Lecture Notes in Artificial Intelligence 2174

Subseries of Lecture Notes in Computer Science
Edited by J. G. Carbonell and J. Siekmann

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Franz Baader Gerhard Brewka
Thomas Eiter (Eds.)

KI 2001: Advances in Artificial Intelligence

Joint German/Austrian Conference on AI
Vienna, Austria, September 19-21, 2001
Proceedings

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editors

Franz Baader
RWTH Aachen, Theoretical Computer Science
Ahornstrasse 55, 52074 Aachen, Germany
E-mail: baader@informatik.rwth-aachen.de

Gerhard Brewka
University of Leipzig, Computer Science Institute, Intelligent Systems Department
Augustusplatz 10-11, 04109 Leipzig, Germany
E-mail: brewka@informatik.uni-leipzig.de

Thomas Eiter
Vienna University of Technology, Institute of Information Systems
Knowledge-Based Systems Group, Favoritenstrasse 11, 1040 Wien, Austria
eiter@kr.tuwien.ac.at

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Advances in artificial intelligence : proceedings / KI 2001, Joint German Austrian Conference on AI, Vienna, Austria, September 19 - 21, 2001. Franz Baader ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; 2174 : Lecture notes in artificial intelligence)
ISBN 3-540-42612-4

CR Subject Classification (1998): I.2

ISBN 3-540-42612-4 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10840436 06/3142 5 4 3 2 1 0

Preface

This volume contains the contributions to the Joint German/Austrian Conference on Artificial Intelligence, KI 2001, which comprises the 24th German and the 9th Austrian Conference on Artificial Intelligence. They are divided into the following categories:

- 2 contributions by invited speakers of the conference;
- 29 accepted technical papers, of which 5 where submitted as application papers and 24 as papers on foundations of AI;
- 4 contributions by participants of the industrial day, during which companies working in the field presented their AI applications.

After a long period of separate meetings, the German and Austrian Societies for Artificial Intelligence, KI and ÖGAI, decided to hold a joint conference in Vienna in 2001. The two societies had previously held one joint conference. This took place in Ottstein, a small town in Lower Austria, in 1986. At that time, the rise of expert system technology had also renewed interest in AI in general, with quite some expectations for future advances regarding the use of AI techniques in applications pervading many areas of our daily life. Since then fifteen years have passed, and we may want to comment, at the beginning of a new century, on the progress that has been made in this direction. Although significant advances in AI research and technology have been made within this period, we are still far from having reached visionary goals such as, for example, the capabilities of HAL, the super computer, in Stanley Kubrick's famous film "2001: A Space Odyssey," let alone Spielberg's interpretation of Kubrick's more recent expectations regarding the future of AI. In this respect, AI is not yet as good as Pinoccio's blue fairy, but easier to find: just go to the annual KI conferences.

The goal of this joint conference was to bring together AI researchers working in academia and in companies, and let them present and discuss their latest research results, both on theoretical foundations as well as on applications; these are the two legs any healthy field needs to stand upon. As we can see from the contributions to this conference, AI appears to be in good shape in this respect. In particular, during the *Industrial Day* of KI 2001, we could observe a number of exciting industrial applications of AI techniques in areas as diverse as configuration, elevator control, supply chain management, and speech recognition. These applications – and many others the general public and sometimes even AI researchers are less aware of – confirm the fact that AI technology has made its way silently into numerous applications (and will certainly continue to do so). In the emerging information society, AI techniques will play a key role for intelligent systems that remain to be built (such as really intelligent search interfaces for the Web). In this respect, the research presented at the conference is encouraging, and makes us confident about the future prospects for and developments in the area.

Following the trend of recent German Conferences on AI, this conference turned out to be not just a local event for AI researchers from Germany and Austria, but an international conference that is of interest to researchers from all over the world. This was reflected in 79 submissions from a total of 22 countries: Algeria, Australia, Austria, China, Cyprus, Finland, France, Germany, Greece, Iran, Israel, Japan, Lithuania, Romania, Slovakia, Spain, Sweden, Switzerland, Taiwan, The Netherlands, UK, and the USA. Of these submissions, 15 were submitted as application papers and 64 as papers on foundations of AI. Of the 29 accepted papers, 3 were quite outstanding in the sense that they clearly obtained the best grades by the reviewers. These papers are grouped under the heading "Selected Papers" in the proceedings and they were presented in a special session (without parallel sessions) at the conference. From these three papers, the program committee selected the paper

> "Approximating Most Specific Concepts in Description Logics with Existential Restrictions" by Ralf Küsters and Ralf Molitor

for the Springer Best Paper Award. Congratulations to the authors for this excellent piece of work!

A large number of people helped to make this conference a success. Our thanks go to the workshop chair (Jürgen Dorn), the industrial chairs (Gerhard Friedrich and Kurt Sundermeyer), the local arrangements chair (Uwe Egly), and all the other people involved in the local organization in Vienna. Special thanks go to Elfriede Nedoma, the conference secretary, whose continuous efforts were vital to the whole enterprise, and to Wolfgang Faber, who maintained the marvelous web-site of the conference and designed posters, folders, and other important items. Thanks Wolfgang, you would be a fabulous artist!

As for the technical program, first and foremost we thank all the authors for submitting their papers to our conference. Secondly, we thank the members of the program committee as well as the additional reviewers who did a tremendous job in writing reviews and participating in the electronic PC meeting. Their effort made it possible to select the best papers from the large number of submissions in a relatively short period of time.

Next, we would like to thank our distinguished invited speakers Hans Kamp (University of Stuttgart), Michael Kearns (Syntek Capital), Raymond Reiter (University of Toronto), and V.S. Subrahmanian (University of Maryland) for kindly accepting our invitation to give a talk at our conference. Furthermore, we thank our tutorialists, Thom Frühwirth (University of Munich) and Stefan Wrobel (University of Magdeburg), for their lectures that made attendants familiar with developments in the areas of constraint handling rules and data mining.

In order to obtain funding for invited speakers and other important events without having to take outrageous conferences fees from the participants, it is vital to obtain support from industrial and other sponsors. Our thanks in this respect go to the following companies: Microsoft, Siemens, Springer-Verlag, and Sysis, and the following institutions and organizations: the European Commission, the Austrian Computer Society (OCG), the Austrian Economic Chamber,

the Austrian Ministry of Transport, Innovation & Technology, the Austrian Ministry of Education, Science and Culture, and the City of Vienna.

Finally, we would like to thank Carsten Lutz (RWTH Aachen) for installing and managing the electronic system (ConfMan) that allowed us to get all submissions via the Internet, and to have a virtual PC meeting. You did a great job! Carsten Lutz together with Ulrike Sattler helped to produce the camera-ready copy of these proceedings.

September 2001

Franz Baader
Gerhard Brewka
Thomas Eiter

KI 2001 Organization

General Chairs

Gerhard Brewka, Leipzig
Thomas Eiter, Wien

Program Chair

Franz Baader, Aachen

Workshop Chair

Jürgen Dorn, Wien

Industrial Chairs

Gerhard Friedrich, Klagenfurt
Kurt Sundermeyer, Berlin

Local Arrangements

Uwe Egly, Wien

Program Committee

Franz Baader, Aachen
Padraig Cunningham, Dublin
Jürgen Dix, Koblenz
Jürgen Dorn, Wien
Didier Dubois, Toulouse
Michael Fisher, Liverpool
Gerhard Friedrich, Klagenfurt
Fausto Giunchiglia, Trento
Horst-Michael Gross, Ilmenau
Andreas Günter, Hamburg
Udo Hahn, Freiburg
Tudor Jebelean, Linz
Jana Koehler, Ebikon
Michael Kohlhase, Pittsburgh
Rudolf Kruse, Magdeburg
Maurizio Lenzerini, Roma
Silvia Miksch, Wien
Bernd Neumann, Hamburg
Michael Richter, Kaiserslautern
Raul Rojas, Berlin
Francesca Rossi, Padova
Ulrike Sattler, Aachen
Jörg Siekmann, Saarbrücken
Peter Struss, München
Michael Thielscher, Dresden
Sebastian Thrun, Pittsburgh
Andrei Voronkov, Manchester
Wolfgang Wahlster, Saarbrücken
Gerhard Widmer, Wien
Stefan Wrobel, Magdeburg

Additional Reviewers

Christoph Benzmüller
Gerd Beuster
Hans-Joachim Böhme
Christian Borgelt
Michael Boronowsky
David Bree
Diego Calvanese
Patrick Clerkin
Marie-Odile Cordier
Anatoli Degtyarev
Clare Dixon
Francesco M. Donini
Paul E. Dunne
Michael Fagan
Markus Färber
Alexander Felfernig
Johannes Fürnkranz
Klaus Fischer
Christian Freksa
Michael Freund
Ulrich Furbach
Leszek Gasieniec
Chiara Ghidini
Alexander Gloye
Henrik Großkreutz
Volker Haarslev
Walther von Hahn
Andreas Herzig
Jochen Hüllen
Susanne Hoche
Ian Horrocks
Lothar Hotz
Ullrich Hustadt
Luca Iocchi
Dietmar Jannach
Aljoscha Klose
Rainer Knauf
Robert Kosara
Mark-A. Krogel
Temur Kutsia
Gerhard Lakemeyer
Jerome Lang
Stefan Langer
Wolfgang Lenski
Carsten Lutz
Jixin Ma
Brian MacNamee
Yves Martin
Johannes Matiasek
Erica Melis
Ralf Möller
Ralf Molitor
Bernhard Nebel
Lourdes Pena Castillo
Johann Petrak
Ian Pratt-Hartmann
Ale Provetti
Jochen Renz
Omar Rifi
Riccardo Rosati
Andrea Schaerf
Marco Schaerf
Tobias Scheffer
Andreas Seyfang
Philippe Smets
Daniele Theseider Duprè
Bernd Thomas
Heiko Timm
Anni-Yasmin Turhan
Helmut Veith
Ubbo Visser
Holger Wache
Thomas Wagner
Toby Walsh
Michael Wessel
Elisabeth Wette-Roch
Emil Weydert
Claus-Peter Wirth
Frank Wolter
Franz Wotawa
Markus Zanker
Gabriele Zenobi

Table of Contents

Computational Game Theory and AI

Michael Kearns

Syntek Capital, 423 W. 55th Street, New York, NY 10019, USA
michael.kearns@syntekcapital.com
http://www.cis.upenn.edu/~ mkearns

Abstract. There has been growing interest in AI and related disciplines in the emerging field of computational game theory. This area revisits the problems and solutions of classical game theory with an explicit emphasis on computational efficiency and scalability. The interest from the AI community arises from several sources, including models and algorithms for multi-agent systems, design of electronic commerce agents, and the study of compact representations for complex environments that permit efficient learning and planning algorithms.
In the talk, I will survey some recent results in computational game theory, and highlight similarities with algorithms, representations and motivation in the AI and machine learning literature. The topics examined will include a simple study of gradient algorithms in general games [1], the application of reinforcement learning algorithms and their generalizations to stochastic games [2], and the introduction of compact graphical models for multi-player games [3,4]. Interesting directions for further work will be discussed.

References

1. S. Singh, M. Kearns, Y. Mansour. Nash Convergence of Gradient Dynamics in General-Sum Games. *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence* (2000), 541–548.
2. M. Kearns, Y. Mansour, S. Singh. Fast Planning in Stochastic Games. *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence* (2000), 309–316.
3. M. Kearns, M. Littman, S. Singh. Graphical Models for Game Theory. *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence* (2001).
4. M. Littman, M. Kearns, S. Singh. An Efficient Exact Algorithm for Singly Connected Graphical Games. Preprint (2001).

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, p. 1, 2001.

Optimal Agent Selection

Fatma Özcan, V.S. Subrahmanian, and Leana Golubchik

Department of Computer Science
University of Maryland, College Park
{fatma,vs,leana}@cs.umd.edu

Abstract. The Internet contains a vast array of sources that provide identical or similar services. When an agent needs to solve a problem, it may split the problem into "subproblems" and find an agent to solve each of the subproblems. Later, it may combine the results of these subproblems to solve the original problem. In this case, the agent is faced with the task of determining to which agents to assign the subproblems. We call this the *agent selection problem* (ASP for short). Solving ASP is complex because it must take into account several different parameters. For instance, different agents might take different amounts of time to process a request. Different agents might provide varying "qualities" of answers. Network latencies associated with different agents might vary. In this paper, we first formalize the agent selection problem and show that it is NP-hard. We then propose a generic cost function that is general enough to take into account the costs of (i) network and server loads, (ii) source computations, and (iii) internal mediator costs. We then develop exact and heuristic based algorithms to solve the agent selection problem.

1 Introduction

There is now a growing body of work on ***software agents***. Several impressive platforms for the creation and deployment of agents have been developed [7,11,12]. In this paper, we focus on the *agent selection problem* (***ASP***) for short).

Suppose we have an agent that provides a set of services. Such an agent may need to access a whole slew of secondary agents to provide its services. For instance, an agent associated with a transportation company may provide shipping services to external parties, as well as provide detailed routing and shipping manifest instructions to its truck drivers. In this case, the transportation agent may obtain routing information from *mapquest* and *mapblast* agents. Likewise, the company's truck drivers may obtain appropriate yellow pages information (e.g., restaurants on the route) from third party servers like *yahoo* and *verizonyellowpages*. Informally put, given a current service S being provided by an agent, the agent selection problem says *how should we select agents to handle the individual atomic services in S so that some performance objective is optimized?*

The answer to this depends on a number of complex factors. First, certain sources may be better for a given service than others — for example, an agent that warehouses data may be "fast" compared to an agent in Indonesia, but the Indonesian source may contain more up to date information. Second, certain agents may have a "heavy" load at some times,

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 2–17, 2001.

but not at others. For instance, it may turn out that *cnnfn* is much more heavily loaded than *financial1.com* at certain times of the day - so at those times using *financial1.com* for stock ticker information may be better. Third, even though *financial1.com* may have lower load at certain times, it may turn out that congestion points on the network outside the control of *financial1.com* cause network access to it to be very slow. Fourth, even though *financial1.com* may have a lighter load than *cnnfn*, its response to requests may include lots of advertisement objects which consume network bandwidth and buffer resources.

All these factors make it clear that given a service S that an agent wishes to provide, assigning agents to handle appropriate parts of S depends not only on the agents themselves, but also on other factors such as the network, extraneous data returned by the sources, freshness of data in the sources, and of course, how well the agents perform their computations.

In this paper, we first formalize the agent selection problem (***ASP***) and show that it is NP-hard. Second, we propose an architecture which uses a "generic" cost function — this is very general and hence, we may plug in different cost models for individual costing of (i) network operations, (ii) source computations and (iii) internal agent computations, and use the cost function to merge them. Third, we provide algorithms that: given a service S will produce a way of assigning agents to the atomic services in S and propose a way of ordering the atomic services of S so as to optimize some performance criterion.

2 Preliminaries

We assume that every agent provides a set of *services* and that agents may be built on top of legacy databases, data structures, and software packages. An agent has a set svc of *primitive service names*. Each service $s \in$ svc has a *signature* specifying the inputs and the outputs of the service. For example, we may have a service called *directions* which has inputs *addr1/addrtype, addr2/addrtype*, which takes two addresses[1] as input and returns a set containing a single list of strings as output (specifying driving directions) if it is possible to drive from the first point to the other, or the empty set if no such route can be generated. It is entirely possible that the same service (e.g., *directions*) may be offered by many different agents.

We assume without loss of generality that all services return, as output, a *set* of objects. As usual, we also assume that each data type τ has an associated *domain* $dom(\tau)$ — a *term* of type τ is either a variable ranging over τ or a member of $dom(\tau)$. If X is a variable of record type τ and $f_i : \tau_i$ is a field of τ, then $X.f_i$ is also a variable. Likewise, if τ_i is itself a record type with field $f_j : \tau_j$, then $X.f_i.f_j$ is also a variable. Variables like $X.f_i$ and $X.f_i.f_j$ are called *path variables* and in this case, their associated *root* variable is X. In general, we use $root(Var)$ to denote the root variable associated with Var. Moreover, if A is a set of variables, then $root(A) = \{root(X) \mid X \in A\}$.

Definition 1 (Service atom). *Suppose s is a service having input signature $(\tau_1, \ldots, \tau_n)$ and t_i is a term of type τ_i for $1 \leq i \leq n$. Furthermore, suppose the output type of s is $\{\tau\}$ and t is a term of type τ. Then: $in(t, s(t_1, \ldots, t_n))$ is a* service atom. *If t is a variable, then it is called the* output variable *of the above service atom.*

[1] *addrtype* may be a record type specifying a number, street name, city, state, zip, and country.

Convention: *In this paper, we use lower case letters to denote constants and upper case letters to denote variables.*

We assume the existence of an agent called *service directory agent* (***SDA***) which may be queried by other agents seeking to know which agents provide which services. There are many ways in which ***SDA*** may be implemented [4,13]. In this paper, we do not go into mechanisms to implement ***SDA***. For the sake of simplicity, we will assume that ***SDA*** has a table of services and agents that offer them. This assumption is not necessary - it may be easily replaced by assuming that an existing implementation of ***SDA*** is available and is used. Table 1 shows a list of services that may be offered by a multiagent application dealing with traffic information.

Table 1. Services offered by the Traffic Information application

Service Name	**Agent(s)**
`directions(address1, address2)`	mapquest, mapblast
`getmap(address)`	mapquest, mapblast
`status(city,highway)`[2]	smartraveler
`status(city,highway)`[3]	etaktraffic
`getinfo(category, name, city, state)`	yahoo,verizonyellowpages
`sqlstring`	facilitiesdb

Users of the traffic information application might want a variety of actions taken on their behalf by agents. Two representative examples are given below.

1. "At least 45 minutes before a scheduled departure, find driving directions to destination x from current location y so that there are no congested roads along the way and print them out."
2. "If the expected arrival at destination x is between 11:30 am and 1:30pm or between 6pm and 8:30 pm, then find the address and the phone number of the closest restaurant to address x and print it out."

A user may request the agent to monitor properties of this type for him. It is important to note that these properties are rules of the form:

$$Head \leftarrow Body$$

where $Head$ is some type of action to be taken, and $Body$ is a condition involving requests (possibly) to third party agents. In the second case above, the request might involve contacting a yellow pages agent (e.g., ***verizonyellowpages*** or ***yahoo***) to obtain a restaurant listing, and then a subsequent request to a map agent (e.g., ***mapquest*** or ***mapblast***) to provide directions to a selected restaurant. The $Body$ part of the above rule is called a *service condition* defined below.

[2] This returns a string, e.g. "no congestion", "slow", etc.

[3] This returns a list of strings, reflecting incidents on the highway

Definition 2 (Comparison atom). *Suppose t_1, t_2 are terms over the same domain. Then $t_1 = t_2$ and $t_1 \not= t_2$ are comparison atoms. In addition, if the domain involved has an associated partial ordering $\leq$, then $t_1 \leq t_2, t_1 < t_2, t_1 \geq t_2, t_1 > t_2$ are comparison atoms where $>, <$ are defined in terms of $\geq, \leq$ and equality in the usual way.*

Definition 3 (Service condition). *A* service condition *is defined inductively as follows: (i) every service atom and comparison atom is a service condition, (ii) if χ_1, χ_2 are service conditions, then so is $\chi_1 \wedge \chi_2$.*

For example, the service condition

$$in(H, getinfo(hospital, nil, bethesda, md)) \wedge in(D, directions(addr1, H.addr))$$

expresses the condition that we want to find a route from `addr1` to a hospital in Bethesda. To evaluate this service condition, one might either use ***Verizonyellowpages*** or ***Yahoo*** to answer the $getinfo()$ condition and either ***Mapquest*** or ***Mapblast*** to compute the $directions()$ condition. Which combination should be chosen depends on a number of parameters including network traffic, historical data on the behavior of the servers, etc.

Even though the ***SDA*** may say that two or more agents (e.g., Mapquest and Mapblast) provide the same service, this does not mean that all these agents provide the same number of answers, nor does it mean that these answers will necessarily be the same. For instance, some sources may be updated in real time, while others might be updated less frequently. Some sources may add objects (e.g., advertisements) to their answers, while others may not. *When two or more agents are listed by the **SDA** as offering the same service, we will assume that the requesting agent is indifferent to any differences in the answers returned by these two agents.*

Throughout the rest of this paper, we assume the existence of some arbitrary, but fixed ***SDA*** that is used by other agents.

3 The Problem

In this section, we describe the technical problem that we solve in this paper. We start by providing some basic notation.

Suppose χ is a service condition. We use $Atoms(\chi)$ to denote the set of all service and comparison atoms in χ. A service condition χ' is said to be a *subconjunct* of χ iff $Atoms(\chi') \subseteq Atoms(\chi)$. A ***service condition partition*** (SCP) of χ is any partition of $Atoms(\chi)$. Intuitively, an SCP of a service condition χ splits χ into subconjuncts $\chi_1, \ldots, \chi_n$ so that each χ_i can be serviced by a single agent.

Definition 4 (Sourced Service Graph). *Consider an agent $\mathcal{A}$ and suppose χ is a service condition. A* sourced service graph (SSG for short) associated with χ *is a directed acyclic graph (V, E) where:*

- *V is a partition of χ and*
- *Every vertex in v is labeled by the name of an agent recognized by **SDA**.*

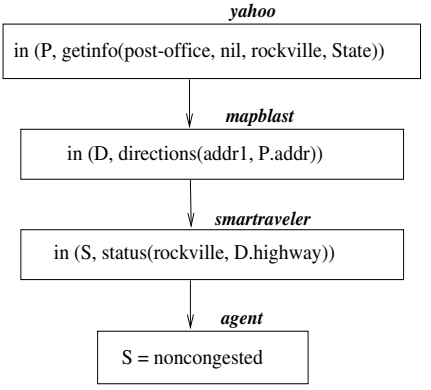


Fig. 1. Example SSG

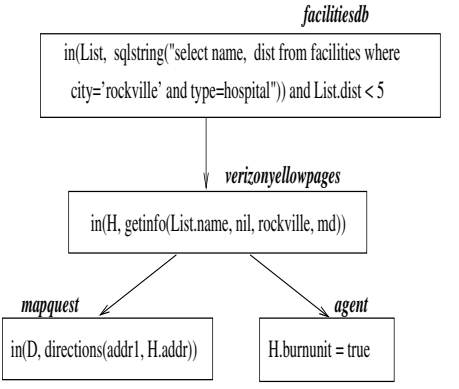


Fig. 2. Example SSG

A sub-SSG *associated with* χ *is a directed acyclic graph* (V, E) *where* $V \subseteq Atoms(\chi)$*, and every vertex is labeled by an agent recognized by* ***SDA***.

Note that for now the above definition poses no restrictions on the edges in E (other than acyclicity). Later, (Definitions 5 and 6) we will impose some constraints on SSGs. Suppose we have a service condition χ and a partition V of χ.

Two examples of sourced service graphs are shown in Figures 1 and 2. It is important to note that a vertex is a set of service atoms and comparison atoms. By taking the conjunction of these atoms, we can associate a service condition $SC(v)$ with the vertex v.

Example 1. The service condition

$$in(P, getinfo(post-office, nil, rockville, State)) \wedge$$
$$in(D, directions(addr1, P.addr)) \wedge$$
$$in(S, status(rockville, D.highway)) \ \wedge \ S = non-congested$$

finds a non-congested route to a post office in Rockville. Figure 1 shows an SSG for it.

The service condition

$$in(List, sqlstring(\text{“}select\ name,\ dist\ from\ facilities\ where$$
$$city = \text{‘}rockville\text{’}\ and\ type = \ hospital\text{”})) \ \wedge List.dist < 5 \wedge$$
$$in(H, getinfo(List.name, nil, rockville, md)) \wedge$$
$$in(D, directions(addr1, H.addr)) \ \wedge \ H.burnunit = true$$

asks for driving directions to a hospital in Rockville which is at most 5 miles from downtown and has a burn unit. The SSG for it is provided in Figure 2. Note that in this SSG *facilitiesdb* agent handles a conjunction. (Agents such as those in the IMPACT system [11] can easily handle conjunctions even if the agents are built on top of data sources which cannot. Hence there is no loss of generality in making this assumption).

It is easy to see that a sourced service graph may not “make sense.” For this to happen, two conditions must be satisfied. First, if an agent labels a vertex, then it must be possible to

execute the service condition labeling that vertex by using just that agent alone. Second, the service condition should be executable, i.e., we should somehow be able to guarantee that all the variables occurring in that service condition which need to be instantiated for it to be executable will in fact be instantiated. For instance, the service condition $in(D, directions(addr1, H.addr))$ can be handled by Mapquest (which provides the "directions" service) but to execute it, $H.addr$ must be instantiated. These two conditions are captured via our notions of a "feasible" sourced service graph and an "evaluable graph" as defined below.

Definition 5 (Feasible SSG). *An SSG (V, E) is* feasible *iff whenever a vertex χ is labeled with agent $\mathcal{A}$, it is the case that every service atom of the form $in(t, s(\ldots))$ is such that s is a service offered by agent $\mathcal{A}$ according to **SDA**.*

Example 2. The SSGs in Figures 1 and 2 are both feasible with respect to the service directory given in Table 1. Both Mapquest and Mapblast offer the *directions* service. Similarly, both Verizonyellowpages and Yahoo provide the service *getinfo*. Finally, Smartraveler offers the *status* service.

Definition 6 (Evaluable SSG). *An SSG (V, E) is* evaluable *iff for every vertex v in V, there is a permutation $\chi_1 \wedge \ldots \wedge \chi_n$ of its atomic constituents such that for all $1 \leq i \leq n$:*

1. *if $\chi_i = in(t, s(t_1, \ldots, t_m))$ and t_r is a (root or path) variable then either:*
   a) *there exists a vertex v' in V such that there is a path from v' to v and there exists either a service atom χ_i' in v' of the form $in(root(t_r), s'(\ldots))$ or a comparison atom χ_i' in v' of the form $root(t_r) = term$ where $term$ is either a variable or constant, or*
   b) *there exists a $j < i$ in v such that either χ_j is of the form $in(root(t_r), s'(\ldots))$ or of the form $root(t_r) = term$ where $term$ is either a variable or constant.*
2. *if χ_i is of the form $t_1 = t_2$, then either at least one of t_1, t_2 is not a variable or either t_1 or t_2 satisfies condition (a) or (b) above;*
3. *if χ_i is of the form $t_1 \underline{op} t_2$ where $\underline{op}$ is one of $\leq, <, \geq, >$, then each of t_1, t_2 must satisfy either condition (a) or (b) above.*

Moreover, a vertex χ in V is evaluable *iff all (root or path) variables t_r appearing in χ satisfy condition (a) or (b) above.*

Example 3. The SSG given in Figure 1 is not evaluable because the atom $in(P, getinfo($ $post-office, nil, rockville, State))$ does not satisfy the above conditions. In particular, there is no way to instantiate the variable $State$. On the other hand, the SSG of Figure 2 is evaluable. This is because every vertex (except for the first one) contains a single atom and each of the atoms satisfies condition (a) of Definition 6 and the first vertex satisfies condition (b) of the definition.

Definition 7 (Cost function). *A* cost function *is a mapping, cost, which takes as input a feasible, evaluable SSG for a service condition χ and returns a real number as output.*

Intuitively, $cost(V, E)$ measures how expensive it is to evaluate the SSG. Due to space restrictions, we do not, in this paper, go into details of what the cost model associated with an agent looks like. The full version of this paper will develop extensive network based cost models for this problem. Suffice it to say that by merely requiring the existence of a cost function that maps SSGs to real numbers, we are allowing ourselves the option of plugging in any kind of cost model whatsoever. Such a cost model will consist of various parameters. These parameters will include ways of measuring: (1) estimated network time, (ii) estimated computation time by remote agents, (iii) estimated cardinality of the set of answers returned by the remote source, (iv) estimated size, in MBytes, of the answer, (v) estimated freshness of the answer, measuring the accuracy of the answer. Based on these parameters, the cost model will combine such values into a single composite value.

Building on top of database query optimization methods, there has been considerable research performed to date into how we may optimize evaluation of a set of service conditions [1,8,3]. All these methods assume the existence of a set of rewrite rules that allow a service condition to be rewritten into many equivalent, but syntactically different forms corresponding to different ways of evaluating the service condition. Given a set RR of rewrite rules for an agent, let $Rew_{RR}(\chi)$ be the set of all possible rewritten versions of service call χ.

We are now ready to define the agent selection problem.

Definition 8 (Agent Selection Problem *(ASP)*). *Suppose $\mathcal{A}$ is an agent and RR is a set of rewrite rules. The agent selection problem (**ASP**) is:*
INPUT: Service condition χ,rewrite rules RR and cost function cost.
OUTPUT: Find a feasible, evaluable SSG (V, E) associated with a member of $Rew_{RR}(\chi)$ such that $cost(V, E)$ is minimized.

Our first result below is that ASP is NP-hard. The proof constructs a reduction of the **sequencing to minimize the weighted completion time** problem [5] to the agent selection problem. This problem can be stated as follows: Given a set T of tasks, a partial order $\prec$ on T, a length $l(t) \in Z^+$, and a weight $w(t) \in Z^+$ for each task $t \in T$, and a positive integer K, is there a one-processor schedule σ for T that obeys the precedence constraints and for which the sum $\sum_{t \in T}(\sigma(t) + l(t)) \times w(t)$ is K or less.

Theorem 1. *The agent selection problem is NP-hard.* □

The rest of this paper consists of two parts. First, in Section 4, we describe an algorithm, **Create-SSG**, which automatically creates a feasible and evaluable SSG for a service condition χ (assuming one exists). This algorithm extends and improves a "safety check" algorithm described in [11]. Then, in Section 5, we develop algorithms to compute a feasible and evaluable SSG which is optimal (w.r.t. a cost function) as well as a heuristic algorithm which computes suboptimal solutions fast.

4 Creating a Feasible and Evaluable SSG

In this section, we develop an algorithm which takes a service condition χ as input and generates a feasible and evaluable SSG for χ (assuming one exists). We use the

following definition to create edges in an SSG. This definition tells us that if a variable which appears as an argument in service atom a_1 is instantiated by another service atom a_2, then a_1 is dependent on a_2 and hence a_2 *may* have to be executed before a_1.

Definition 9 (Dependent service conditions). *A service condition χ_j is said to be* dependent on *χ_i if and only if the following holds:*

1. ***Case 1:*** *χ_i is a service atom of the form $in(X_i, s(t_1, \ldots, t_m))$.*
   a) *If χ_j is a service atom of the form $in(X_j, s'(t_1, \ldots, t_n))$ then $\exists j (1 \leq j \leq n)$ s.t. $root(t_j) \in root(X_i)$.*
   b) *If χ_j is a comparison atom of the form $t_1 = t_2$, then either t_1 is a variable and $root(t_1) \in root(X_i)$ or t_2 is a variable and $root(t_2) \in root(X_i)$.*
   c) *If χ_j is a comparison atom of the form t_1 $\underline{op}$ t_2, where $\underline{op}$ is one of $\leq, <, \geq, >$, then either t_1 is a variable and $t_1 \in root(\overline{X_i})$ or t_2 is a variable and $root(t_2) \in root(X_i)$, or both.*
2. ***Case 2:*** *χ_i is a comparison atom of the form t_1 $\underline{op}$ t_2*[4]*.*
   a) *If χ_j is a service atom of the form $in(X_j, s(t_1, \ldots, t_n))$ then $\exists j (1 \leq j \leq n)$ s.t. $root(t_j) \in root(var(\chi_i))$.*
   b) *If χ_j is a comparison atom of the form $t_3 = t_4$, then either t_3 is a variable and $root(t_3) \in root(var(\chi_i))$ or t_4 is a variable and $root(t_4) \in root(var(\chi_i))$.*
   c) *If χ_j is a comparison atom of the form t_3 $\underline{op}$ t_4, where $\underline{op}$ is one of $\leq, <, \geq, >$, then either t_3 is a variable and $root(t_3) \in root(var(\overline{\chi_i}))$ or t_4 is a variable and $root(t_4) \in root(var(\chi_i))$ or both.*

The **Create-SSG** algorithm is given in Figure 3. The **Create-SSG** algorithm creates an SSG where each vertex contains only one atom and each vertex is labeled with an agent that can execute the atom. The algorithm first computes the set of atoms (Ok) which do not depend on any other atoms. It keeps track of variables that are instantiated in the set Var, and uses this set to determine which other atoms become evaluable. At each iteration of the while loop, the algorithm identifies the set of atoms that become evaluable, inserts the set of variables which are instantiated by those atoms, and creates the minimal number of edges implied by the dependency relations.

Theorem 2. *The **Create-SSG** algorithm generates a feasible and evaluable SSG, if one exists.* □

We have implemented the **Create-ssg** algorithm on a Sun Ultra1 machine with 320 MB memory running Solaris 2.6. We generated SDAs containing 10 services, each having 2 agents that provide that service. As the **Create-ssg** algorithm randomly picks one agent to label each vertex, we used this setting in all the experiments. We ran two sets of experiments. In the first set, we kept the number of dependencies constant and varied the number of conjuncts from 5 to 40. We repeated the same experiments when 10, 20, 30, and 40 dependencies are present. For each combination of number of dependencies and conjuncts, we created 1000 service conditions and recorded the average running time. Figure 4 shows the results. As seen from the figure, the execution time increases linearly with the number of conjuncts. The **Create-SSG** algorithm is extremely fast,

[4] if $\underline{op}$ is "=", then at most one of t_1 or t_2 is a variable.

Create-SSG(χ)
/* **Input**: $\chi : \chi_1 \wedge \chi_2 \wedge \ldots \wedge \chi_n$ */
/* **Output** : an evaluable SSG (V,E) if one exists, NIL otherwise */

(1) L:= $\{\chi_1, \chi_2, .., \chi_n\}$; (2) $L' := \emptyset$; (3) $Var := \emptyset$; (4) $E := \emptyset$;
(5) $V := \{\chi_i \mid 1 \leq i \leq n\}$;
(6) **forall** $\chi_i \in V$
(7) label χ_i with s, s.t. $s \in SD[\chi_i].list$
(8) $Ok := \{\chi_i \mid \chi_i$ is either of the form in(X, s(args)) where args are ground, or of the form
(9) $t_1 = t_2$ where at most one of t_1 or t_2 is a *root* variable, or of the form t_1 $\underline{op}$ t_2,
(10) where $\underline{op} \in \{<, >, \leq, \geq\}$ and both t_1 and t_2 are constants $\}$;
(11) $Var := Var \cup \{root(X_i) \mid (in(X_i, s(args)) \in Ok \; or \; X_i = constant \in Ok)$
(12) $and \; X_i = root(X_i)\}$;
(13) L:= $L - Ok$; L':=$L' \cup Ok$;
(14) **while** (L is not empty) **do**
(15) $\Psi := \{\chi_i \mid \chi_i \in L$ and all root variables in χ_i are in $Var\}$;
(16) **if** card(Ψ) = 0 **then Return** NIL;
(17) **else** $Var := Var \cup \{root(X_i) \mid (in(X_i, s(args)) \in \Psi \; or$
(18) $X_i = constant \in \Psi) \; and \; X_i = root(X_i)\}$;
(19) **forall** pairs $\langle \chi_i, \chi_j \rangle$, $\chi_j \in \Psi$, s.t. χ_j is dependent on $\chi_i \in L'$
(20) $E := E \cup \{\langle \chi_i, \chi_j \rangle\}$;
(21) $L := L - \Psi$; $L' := L' \cup \Psi$;
(22) **end(while)**
(23) **Return** (V, E);
End-Algorithm

Fig. 3. Create-SSG Algorithm

taking less than 30 milliseconds for service conditions involving 40 conjunctions and 25 dependencies.

In the second set of experiments, we kept the number of conjuncts constant, and varied the number of dependencies from 10 to 50. We ran four experiments with 10, 20, 30, and 40 conjuncts. Again, we generated 1000 service conditions for each combination and used the average running time. The results are given in Figure 5. Again, the execution time increases linearly with the number of dependencies.

5 Agent Selection Algorithms

In this section, we describe two algorithms that take a service condition χ as input and produce a sourced service graph as output. The first algorithm, A^***-SSG**, is an A^*-based algorithm which finds an optimal solution. The A^***-SSG** algorithm maintains nodes n with the following fields: a (sub-)SSG, a cost function value $g(n)$ and a heuristic function value $h(n)$. The algorithm starts out with an empty graph and builds evaluable and feasible sub-SSGs by considering one atom at a time. The algorithm expands one node at a time and maintains a list OPEN of nodes ordered in ascending order of $g(n) + h(n)$.

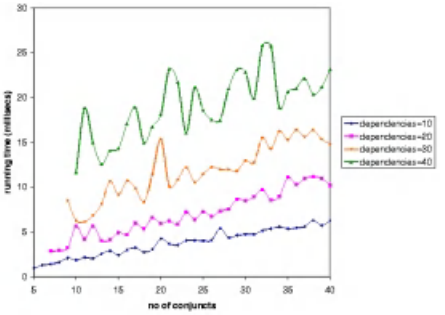


Fig. 4. Execution Time of **Create-SSG** (constant no of dependencies)

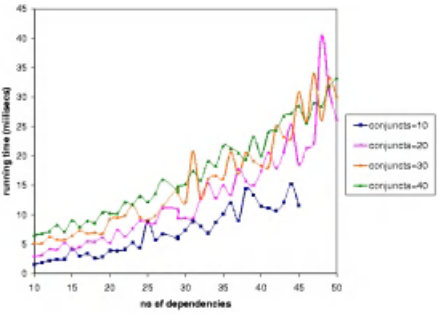


Fig. 5. Execution Time of **Create-SSG** (constant no of conjuncts)

At each stage, the A^***-SSG** algorithm expands the node with the smallest $g(n) + h(n)$ value. In order to define the notion of an expansion, we first define a "semi-expansion."

Definition 10 (Semi-Expansion). $G' = (V', E')$ semi-expands $G = (V, E)$ *with atom* χ_i *if*

1. $\chi_i \in Atoms(\chi) - Atoms(G)$, *and*
2. G' *is feasible and evaluable, and*
3. $E \subseteq E'$, *and*
4. *either* $V' = V \cup \{\chi_i\}$, *and* χ_i *is labeled with some agent* a *which provides the service referenced in* χ *according to* ***SDA***.
   or $V' = (V - \{v\}) \cup \{v'\}$, *for some* $v \in V$ *such that* $v' = v \wedge \chi_i$, *and* $label(v)$ *is an agent that provides the service referenced in* χ *according to* ***SDA***.

$G' = (V', E')$ *minimally semi-expands* G *with* χ_i *iff*

1. G' *expands* G *with* χ_i, *and*
2. $\nexists G''(V'', E'')$ *such that* G'' *expands* G *with* χ_i *and* $E'' \subset E'$.

Finally, $SemiExpansions(G) = \{G' \mid \exists \chi_i \in (Atoms(\chi) - Atoms(G))$ *such that* G' *minimally semi-expands* G *with* $\chi_i\}$.

A minimal expansion intuitively adds a new atom into G without removing any existing edges in E — however, some new edges might be added and/or some existing vertices might be expanded.

Definition 11 (Expansion). *Suppose* $G = (V, E)$ *is as above and* RR *is some set of rewrite rules.* $G' = (V', E')$ expands $G = (V, E)$ *with atom* χ_i *if* G' *is a semi-expansion w.r.t.* χ_i *of* some rewriting of *the service condition* $\bigwedge_{\chi_j \in Atoms(G)} \chi_j$) *using the rewrite rules in* RR.

We use $Expansions(G)$ *to denote the set of all expansions of* G.

It is important to note that there may be many elements in $Expansions(G)$. We are now ready to define the cost function $g(n)$ and the heuristic function $h(n)$ associated with our A^*-SSG algorithm.

Definition 12 (Functions $g(n)$ and $h(n)$). *Let $G = (V, E)$ be the sub-SSG associated with a node of the A^*-SSG algorithm, then $g(n)$ and $h(n)$ are defined as follows:*

$$g(n) = cost(V, E)$$
$$h(n) = min_{G'=(V',E') \in Expansion(G)}(cost(V', E')) - cost(V, E)$$

The heuristic function h gives the minimum cost increment when we include one more (service or comparison) atom in the current sub-SSG.

```
A*-SSG(χ, SD)
/* Input: a service condition χ */
/* Output: an evaluable SSG of χ if one exists, NIL otherwise */

OPEN := ∅
Ground := {χ_i | χ_i is either of the form in(X, s(args)) where args are ground, or of
              the form t_1 = t_2 where at most one of t_1 or t_2 is a root variable, or of
              the form t_1 op t_2, (op ∈ {<, >, ≤, ≥}) and both t_1 and t_2 are constants };
forall χ_i ∈ Ground do
   forall agents s that offer χ_i
      create a sub-SSG G with one vertex v containing χ_i and label s
      create a new node n with G
      insert n into OPEN
sort OPEN in increasing order of f(n)
while (OPEN ≠ ∅) do
   n := OPEN.head
   delete n from OPEN
   if n is a goal node then Return(n.G)
   else /* expand and generate children */
      insert all nodes n' ∈ Expansion(n.G) into OPEN
end(while)
Return (NIL)
End-Algorithm
```

Fig. 6. The A^*-SSG Algorithm

The A^***-SSG** algorithm is given in Figure 6. To create *evaluable* sub-SSGs, the A^*-**SSG** algorithm first computes the set of service atoms, $Ground$, which do not depend on any other service atoms, and hence are evaluable right away. It creates sub-SSGs containing just one atom from $Ground$. Moreover, the **Expansions** function only creates evaluable and feasible sub-SSGs. If the while loop of the algorithm terminates, this implies that no goal state was reached and hence there is no solution. A node n is a goal state for χ if $n.G$ is an evaluable and feasible SSG of χ. Note that if the heuristic function

h used by the A^* algorithm is admissible, then the algorithm is guaranteed to find the optimum solution [9]. The heuristic h is admissible iff $h(n) \leq h^*(n)$ where $h^*(n)$ is the lowest actual cost of getting to a goal node from node n. The following result shows that our heuristic is admissible.

Theorem 3 (Admissibility of h). *For all nodes n, $h(n) \leq h^*(n)$. Hence, the A^*-SSG algorithm finds an optimal solution.* □

5.1 Heuristic Algorithm

In this section, we describe a greedy algorithm that uses heuristics to solve the agent selection problem. As the algorithm is a greedy one, it is not guaranteed to find an optimal solution. The algorithm starts out by creating a non-optimal feasible and evaluable initial solution. It iteratively improves the initial solution by applying a series of SSG transformations: *Merge* and *Relabel* that we introduce below. *Merge* merges two vertices of the SSG into one, whereas *Relabel* changes the label of a vertex. We first define these transformations, and then we describe our heuristic.

Definition 13 (Merge). *The **Merge** transformation takes an SSG $G = (V, E)$ and two vertices χ_i and χ_j as input and generates another SSG $G' = (V', E')$ as output such that*

1. *If $label(\chi_i) = label(\chi_j)$ and either there exists an edge $e \in E$ between χ_i and χ_j, or there exists no path between χ_i and χ_j, then*
   - *$V' = (V - \{\chi_i, \chi_j\}) \cup \{\chi_i \wedge \chi_j\}$, and*
   - *$E' = E - \{(\chi_l, \chi_k) \mid \chi_l \in \{\chi_j, \chi_i\}$ or $\chi_k \in \{\chi_j, \chi_i\}\} \cup \{(\chi_i \wedge \chi_j, \chi_k)\} \cup \{(\chi_l, \chi_i \wedge \chi_j)\}$*
2. *Otherwise,*
   - *$V' = V$ and $E' = E$*

The *Merge* transformation merges two vertices into one by taking the conjunction of the service conditions in those two vertices. It does so only if the two vertices are labeled with the same agent. It deletes all edges that are between other vertices χ_k and either χ_i or χ_j and inserts new edges that are now between that vertex and the merged vertex $\chi_i \wedge \chi_j$. The following example illustrates how the **Merge** transformation is applied.

Example 4. Let $\chi_1 = in(X, sc_1(a))$, $\chi_2 = in(Y, sc_2(X))$, $\chi_3 = in(Z, sc_3(b))$ and $\chi_4 = in(W, sc_4(Y, Z))$. Suppose $\chi = \chi_1 \wedge \chi_2 \wedge \chi_3 \wedge \chi_4$, and suppose we know that the following agents provide the services $sc_1, \ldots, sc_4$:

$$\begin{array}{l} sc_1 : agent1, agent2 \\ sc_2 : agent2, agent5 \\ sc_3 : agent3 \\ sc_4 : agent1, agent4 \end{array}.$$

Consider the feasible and evaluable SSG of Figure 7. If we apply the **Merge** transformation $Merge(G, \chi_1, \chi_2)$, we get the SSG G' given in Figure 8. We can apply the **Merge** transformation because χ_1 and χ_2 are labeled with the same agent, $agent2$, and there exists an edge between them.

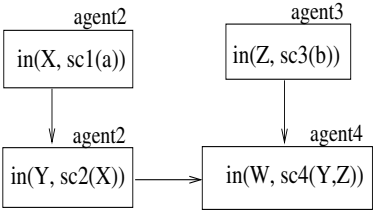


Fig. 7. SSG before **Merge**

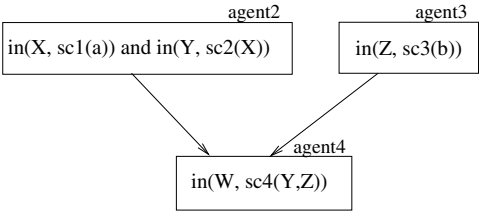


Fig. 8. SSG after **Merge**

Proposition 1. *If G is evaluable and feasible, then so is $Merge(G, \chi_i, \chi_j)$.* □

Definition 14 (Relabel). *The **Relabel** transformation takes an SSG $G = (V, E)$ and a vertex χ_i as input and generates an SSG $G' = (V', E')$ as output such that*

1. *$V' = V$ and $E' = E$*
2. *$label(\chi_i) = s_1 \in G$, and $label(\chi_i) = s_2 \in G'$ for one $\chi_i \in V$, and $s_1, s_2 \in SD[\chi_i].list$.*

Proposition 2. *If G is evaluable and feasible, then so is $Relabel(G, \chi)$.* □

The order in which these transformations are applied can be significant. For example, suppose we have two vertices v_1 and v_2 in an SSG. Suppose v_1 is labeled with agent $\mathcal{A}_1$ and v_2 is labeled with $\mathcal{A}_2$. Further, suppose a third agent $\mathcal{A}_3$ is able to process the service conditions in both v_1 and v_2. If we first apply the **Relabel** transformation to both v_1 and v_2 and change both labels to $\mathcal{A}_3$, we can also apply the **Merge** transformation. In order to catch such possibilities, we need to consider all possible transformations in all possible orders. However, as the size of the SSG increases such a search space becomes too large to handle. Hence, instead of creating all possible SSGs, we propose a heuristic-based algorithm.

The **Greedy-Order** algorithm is provided in Figure 9. It first creates a feasible and evaluable SSG by using the **Create-SSG** algorithm. It then creates a queue of vertices, sorted in decreasing order of cost. The algorithm examines one vertex at a time, and applies all possible transformation to this vertex. It chooses the best cost SSG among all the resulting SSGs, and sets this SSG as the current SSG. The algorithm terminates when it exhausts the vertices in the queue.

6 Related Work

To date, most existing work on what part of a service request should be assigned to which agent has been based on "matchmaking" efforts. In matchmaking efforts, agents advertise their services, and matchmakers match an agent requesting a service with one (or more) that provides it. Four of the best known examples of this class of work are given below.

Greedy-Order(χ, SD)
/* **Input**: a service condition χ. */
/* **Output**: an evaluable SSG of χ */

$G(V, E)$:= Create-SSG(χ);
$currSSG := G$; $currCost := cost(V, E)$;
create QUEUE and insert vertices of G into QUEUE;
order vertices in QUEUE in decreasing order of cost;
while (QUEUE $\neq emptyset$) **do**
  χ_i := QUEUE.dequeue;
  $minSSG := currSSG$; $minCost := currCost$;
  forall (χ_j s.t. $G' = \mathbf{Merge}(currSSG, \chi_i, \chi_j) \neq G$) **do**
    $G'(V', E') := \mathbf{Merge}(currSSG, \chi_i, \chi_j)$;
    if ($cost(V', E') < minCost$) **then**
      $minSSG := G'$; $minCost := cost(V', E')$;
  $currSSG := minSSG$; $currCost := minCost$;
  forall ($s \in SD[\chi_i].list$) **do**
    $G'(V', E') := \mathbf{Relabel}(currSSG, \chi_i, s)$;
    if ($cost(V', E') < minCost$) **then**
      $minSSG := G'$; $minCost := cost(V', E')$;
  $currSSG := minSSG$; $currCost := minCost$;
end(while)
Return($currSSG$);
End-Algorithm

Fig. 9. The **Greedy-Order** Algorithm

Kuokka and Harada[6] present the SHADE and COINS systems for matchmaking. SHADE uses logical rules to support matchmaking – the logic used is a subset of KIF and is very expressive. In contrast, COINS assumes that a message is a document (represented by a weighted term vector) and retrieves the "most similar" advertised services using the SMART algorithm of Salton[10].

Decker, Sycara, and Williamson[4] present matchmakers that store capability advertisements of different agents. They look for *exact* matches between requested services and retrieved services, and concentrate their efforts on architectures that support load balancing and protection of privacy of different agents.

Arisha et. al. [2] develop matchmakers in which each agent advertises its services via an HTML style language. A metric space is defined on this language, and the job of matchmaking is now reduced to a nearest neighbor search on such a metric space.

Vassalos and Papakonstantinou [13] develop a rule based matchmaking language. In this framework, when an agent sends a request for a service, the rules are used to match the requested service with an agent that provides it.

In contrast to the above efforts, our work builds on top of a "matchmaker." Matchmakers may be used to implement the ***SDA*** described in our framework. However, we make novel use of the ***SDA***. Specifically, we use costs to determine, of many possible

agents that could satisfy a service request, which ones should actually be assigned to the job. This determination not only takes into account possible methods to rewrite service requests, but also different possible assignments of agents to sub-requests. This yields two dimensions of complexity: rewriting of service conditions as well as assignment of agents to perform the task.

7 Conclusions

Agents are playing an increasingly important role in a vast variety of applications. These applications range from personalized marketing agents, personalized presentation agents, agents to integrate heterogeneous databases and software packages, agents for supply chain management, agents for battlefield applications, and many, many others. This explosion of agent research has led to the development of several excellent agent development systems including, but not limited to systems such as Aglets from IBM [7], IMPACT [11], and Retsina [12].

We believe that for agents to be effective, they must be "small." Small agents typically perform a small number of clearly articulated, well defined tasks (even though these tasks may involve accessing a huge amount of data). Verifying a small program is much easier than a large program. In addition, small programs are easier to modify and maintain. Additionally, should these programs be mobile, then their requirements on a host machine are small (i.e. they have a small footprint on the host). By having a large number of small agents collaborate with one another, we may build large agent applications.

In such situations, we will often have multiple agents that provide similar services. This situation is already widespread on the Internet. There are thousands of news sources, and hundreds of yellow page information sources. Sources on books and videos abound. When an agent wishes to access a service that it does not already provide, it must (i) find other agents that provide the desired services, and (ii) determine which of those agents to actually use to obtain the desired service.

Problem (i) above has already been solved by "matchmaking" technology described in the preceding section. The goal of this paper is to address problem (ii). We believe that performance issues must drive the choice made in problem (ii). Thus, in this paper, we have proposed how an agent can make choices about how to execute a service condition and which agents should be selected to execute appropriate parts of (a perhaps rewritten) service condition. This problem, which we call the Agent Selection Problem (***ASP*** for short) is proved in this paper to be NP-hard. We then develop two algorithms to solve this problem — an algorithm that computes an optimal solution, building on top of the well known $A^\star$ algorithm, and a greedy algorithm.

Acknowledgements. This work was supported by the Army Research Lab under contract number DAAL0197K0135, the Army Research Office under grant number DAAD190010484, by DARPA/RL contract number F306029910552 and by NSF grant CCR-98-96232.

References

1. S. Adali, K. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 137–148, Montreal, Canada, June 1996.
2. K. Arisha, F. Ozcan, R. Ross, V. S. Subrahmanian, T. Eiter, and S. Kraus. Impact: A platform for collaborating agents. *IEEE Intelligent Systems*, 14:64–72, March/April 1999.
3. S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 34–43, Seattle, Washington, USA, June 1998.
4. K. Decker, K. Sycara, and M. Williamson. Middle agents for the internet. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 578–583, Nagoya, Japan, 1997.
5. M. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York City, USA, 1979.
6. D. Kuokka and L. Harada. Integrating information via matchmaking. *Journal of Intelligent Informations Systems*, 6(3):261–279, 1996.
7. D. Lange and M. Osjima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.
8. A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedins of the 22nd International Conference on Very Large Databases (VLDB)*, pages 251–262, Bombay, India, September 1996.
9. N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, 1986.
10. G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
11. V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogenous Active Agents*. MIT Press, Cambridge, Massachusetts, USA, June 2000.
12. K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The retsina mas infrastructure, 2000. Submitted to JAAMS.
13. V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *Journal of Logic Programming*, 43:75–122, 2000.

Towards First-Order Temporal Resolution

Anatoli Degtyarev and Michael Fisher

Logic and Computation Group, Department of Computer Science
University of Liverpool, Liverpool L69 7ZF, U.K.
{A.Degtyarev,M.Fisher}@csc.liv.ac.uk

Abstract. In this paper we show how to extend clausal temporal resolution to the ground eventuality fragment of monodic first-order temporal logic, which has recently been introduced by Hodkinson, Wolter and Zakharyaschev. While a finite Hilbert-like axiomatization of complete monodic first order temporal logic was developed by Wolter and Zakharyaschev, we propose a temporal resolution-based proof system which reduces the satisfiability problem for ground eventuality monodic first-order temporal formulae to the satisfiability problem for formulae of classical first-order logic.

1 Introduction

We consider the first-order temporal logic over the natural numbers $TL(\mathbb{N})$ in a first-order temporal language $\mathcal{TL}$. The language $\mathcal{TL}$ is constructed in the standard way (see i.e. [Fis97,HWZ00]) from a classical (non-temporal) first-order language $\mathcal{L}$ and a set of future-time temporal operators '$\Diamond$' (*sometime*), '$\Box$' (*always*), '$\bigcirc$' (*in the next moment*), '$\mathcal{U}$'(*until*) and '$\mathcal{W}$' (*unless*, or *weak until*). Here, $\mathcal{L}$ does not contain equality or functional symbols.

Formulae in $\mathcal{TL}$ are interpreted in *first-order temporal structures* of the form $\mathfrak{M} = \langle D, \mathcal{I} \rangle$, where D is a non-empty set, the *domain* of $\mathfrak{M}$, and $\mathcal{I}$ is a function associating with every moment of time $n \in \mathbb{N}$ an interpretation of predicate and constant symbols of $\mathcal{L}$ over D. First-order (nontemporal) structures corresponding to each point of time n will be denoted by $\mathfrak{M}_n = \langle D, I_n \rangle$ where $I_n = \mathcal{I}(n)$. Intuitively, the interpretations of $\mathcal{TL}$-formulae are sequences of *worlds* such as $\mathfrak{M}_0, \mathfrak{M}_1, \ldots, \mathfrak{M}_n \ldots$. An *assignment* in D is a function $\mathfrak{a}$ from the set $\mathcal{L}_\mathbf{v}$ of individual variables of $\mathcal{L}$ to D. We require that (individual) variables and constants of $\mathcal{TL}$ are *rigid*, that is neither assignments nor interpretations of constants depend on worlds.

The *truth-relation* $\mathfrak{M}_n \models^\mathfrak{a} \varphi$ (or simply $n \models^\mathfrak{a} \varphi$, if $\mathfrak{M}$ is understood) in the structure $\mathfrak{M}$ for the assignment $\mathfrak{a}$ is defined inductively in usual way under the following semantics of temporal operators:

$$
\begin{array}{lll}
n \models^\mathfrak{a} \bigcirc \varphi & \text{iff} & n+1 \models^\mathfrak{a} \varphi; \\
n \models^\mathfrak{a} \Diamond \varphi & \text{iff} & \text{there exists a } m \geq n \text{ such that } m \models^\mathfrak{a} \varphi; \\
n \models^\mathfrak{a} \Box \varphi & \text{iff} & m \models^\mathfrak{a} \varphi \text{ for all } m \geq n; \\
n \models^\mathfrak{a} \varphi \,\mathcal{U}\, \psi & \text{iff} & \text{there exists a } m \geq n \text{ such that } m \models^\mathfrak{a} \psi \text{ and} \\
 & & \text{for every } k \in \mathbb{N}, \text{ if } n \leq k < m \text{ then } k \models^\mathfrak{a} \varphi : \\
n \models^\mathfrak{a} \varphi \,\mathcal{W}\, \psi & \text{iff} & n \models^\mathfrak{a} \varphi \,\mathcal{U}\, \psi \text{ or } n \models^\mathfrak{a} \Box \varphi.
\end{array}
$$

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 18–32, 2001.

A formula φ is said to be *satisfiable* if there is a first-order structure $\mathfrak{M}$ and an assignment $\mathfrak{a}$ such that $\mathfrak{M}_0 \models^{\mathfrak{a}} \varphi$. If $\mathfrak{M}_0 \models^{\mathfrak{a}} \varphi$ for every structure $\mathfrak{M}$ and for all assignments, then φ is said to be *valid*. Note that formulae here are interpreted in the initial world $\mathfrak{M}_0$; that is an alternative but equivalent definition to the one used in [HWZ00].

2 Divided Separated Normal Form

Our method works on temporal formulae transformed into a normal form. This normal form follows the spirit of Separated Normal Form (SNF) [Fis91,FDP01] and First-Order Separated Normal Form (SNF$_f$) [Fis92,Fis97]. However, we go even further.

One of the main aims realized in SNF/SNF$_f$ was inspired by Gabbay's separation result [Gab87]. In accordance with this aim, formulae in SNF/SNF$_f$ comprise implications with present-time formulae on the left-hand side and (present or) future formulae on the right-hand side. The transformation into the separated form is based upon the well-known *renaming* technique [PG86], which preserves satisfiability and admits the extension to temporal logic in (Renaming Theorems [Fis97]).

Another intention was to reduce most of the temporal operators to a core set. This concerns the removal of temporal operators represented as *maximal* fixpoints, i.e. $\Box$ and $\mathcal{W}$ (Maximal Fixpoint Removal Theorems [Fis97]). Note that the $\mathcal{U}$ operator can be represented as a combination of operators based upon maximal fixpoints and the $\Diamond$ operator (which is retained within SNF/SNF$_f$). This transformation is based upon the simulation of fixpoints using QPTL [Wol82].

Now we add one additional aim, namely to divide the temporal part of a formula from its (classical) first-order part in such way that the temporal part is as simple as possible. The modified normal form is called Divided Separated Normal Form or DSNF for short. A Divided SNF problem is a triple $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ where $\mathcal{S}$ and $\mathcal{U}$ are the *universal part* and the *initial part*, respectively, given by finite sets of nontemporal first-order formulae (that is, without temporal operators), and $\mathcal{T}$ is the *temporal part* given by a finite set of *temporal clauses*. All formulae are written in $\mathcal{L}$ extended by a set of predicate and propositional symbols. A temporal clause has one of the following forms:

$$P(\bar{x}) \Rightarrow \bigcirc \bigwedge_{i=1}^{n} Q_i(\bar{x}) \quad \textit{(predicate step clause)},$$

$$p \Rightarrow \bigcirc \bigwedge_{j=1}^{m} q_j \quad \textit{(proposition step clause)},$$

$$P(\bar{x}) \Rightarrow \bigcirc \Diamond Q(\bar{x}) \quad \textit{(predicate eventuality clause)},$$

$$p \Rightarrow \bigcirc \Diamond q \quad \textit{(proposition eventuality clause)}$$

where P, Q, Q_i are predicate symbols, p, q, q_j are propositional symbols, and $\Rightarrow$ is a substitute for implication. Sometimes temporal clauses are called *temporal rules* to make distinctions between their left- and right-hand sides. Without loss of generality we suppose that there are no two different temporal step rules with the same left-hand sides and there are no two different eventuality rules with the same right-hand sides. An atom $Q(\bar{x})$ or q from the right-hand side of an eventuality rule is called an *eventuality atom*.

We call examples of DSNF *temporal problems*. The semantics of a temporal problem is defined under the supposition that the universal and temporal parts are closed by the outermost prefixes $\Box\forall$, the initial part is closed only by universal quantifiers. In what follows we will not distinguish between a finite set of formulae $\mathcal{X}$ and the conjunction $\bigwedge\mathcal{X}$ of formulae in it. Thus the temporal formula corresponding to a temporal problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is $(\Box\forall\mathcal{U}) \wedge \forall\mathcal{S} \wedge (\Box\forall\mathcal{T})$.

So, when we consider the satisfiability or the validity of a temporal problem we implicitly mean the corresponding formula, as above.

Given the results about the renaming of subformulae and the removal of temporal operators mentioned above, we can state the general theorem about translation into DSNF as follows.

Theorem 1. *Any first-order temporal formula φ in $\mathcal{TL}$ can be translated into a temporal problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ (i.e. DSNF of φ) in a language $\mathcal{TL}' \supseteq \mathcal{TL}$ extended by new propositional and predicate symbols such that φ is satisfiable if, and only if, $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is satisfiable.*

For any formula φ its DSNF representation $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ can be constructed in polynomial time in the length of φ. (As a whole the transformation of φ into $< \mathcal{S}, \mathcal{U}, \mathcal{T} >$ is similar to the familiar depth-reducing reductions of first-order formulae via the introduction of new names.)

Example 1. Let us consider the following formula: $\varphi = (\exists x \Box\neg Q(x)) \wedge (\bigcirc \exists y Q(y))$. After transformation φ to a normal form, we get the following temporal problem:

$$\mathcal{S} = \left\{ \begin{array}{l} s1.\ p_1 \\ s2.\ p_3 \end{array} \right\}, \quad \mathcal{T} = \left\{ \begin{array}{l} t1.\ p_1 \Rightarrow \bigcirc p_2, \\ t2.\ P_1(x) \Rightarrow \bigcirc (P_2(x) \wedge P_1(x)) \end{array} \right\},$$

$$\mathcal{U}_0 = \left\{ \begin{array}{lll} u1.\ P_3(x) \supset P_2(x), & u3.\ P_2(x) \supset \neg Q(x), & u5.\ p_3 \supset \exists x P_3(x) \\ u2.\ P_3(x) \supset P_1(x), & u4.\ p_2 \supset \exists y Q(y), & \end{array} \right\}.$$

3 The Monodic Fragment and Merged Temporal Step Rules

Following [HWZ00] we consider the set of all $\mathcal{TL}$-formulae φ such that any subformula of φ of the form $\Diamond\psi$, $\Box\psi$, $\bigcirc\psi$, $\psi_1 \,\mathcal{U}\, \psi_2$, $\psi_1 \,\mathcal{W}\, \psi_2$ has at most one free variable. Such formulae are called *monodic*, and the set of monodic $\mathcal{L}$-formulae is denoted by $\mathcal{T}_1\mathcal{L}$. In spite of its relative narrowness the monodic fragment provides a way for quite realistic applications. For example, temporal extensions of the spatial formalism RCC-8 [Wol00] lie within the monodic fragment. Another example is the verification of properties of relational transducers for electronic commerce [AVFY00] which are expressed in the monodic language again.

The decidability of $\mathcal{T}_1\mathcal{L}$ was proved in [HWZ00] while, in [WZ01], a finite Hilbert-style axiomatization of the monodic fragment of $TL(\mathbb{N})$ has been constructed. However no deduction-based decision procedure for this class has yet been proposed.

The notion 'monodic' is transferred from temporal formulae to temporal problems as follows. A problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is *monodic problem* if all predicates occurring in its temporal part $\mathcal{T}$ are monadic. Every monodic formula is translated into DSNF given

by the monodic problem. In Example 1 both the formula φ and its DSNF problem are monodic.

The key role in propositional temporal resolution is played by so-called *merged step clauses* [FDP01]. In the case of the monodic fragment, we can define an analogue of propositional merged step clauses and so formulate for monodic problems a calculus which is analogous to the propositional temporal resolution calculus (up to replacing the propositional merged clauses by the first-order merged clauses defined below) such that this calculus is complete for the so-called *ground eventuality monodic fragment* defined in the next section.

Next we introduce the notions of colour schemes and constant distributions. Let $\mathcal{P} =< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ be a temporal problem. Let C be the set of constants occurring in $\mathcal{P}$. Let $\mathcal{T}^{\mathbf{P}} = \{P_i(x) \Rightarrow \bigcirc R_i(x), \mid 1 \le i \le K\}$ and $\mathcal{T}^{\mathbf{p}} = \{p_j \Rightarrow \bigcirc r_j \mid 1 \le j \le k\}$ be the sets of all predicate step rules and all propositional step rules of $\mathcal{T}$, respectively. It is supposed that $K \ge 0$ and $k \ge 0$; if $K = 0$ ($k = 0$) it means that the set $\mathcal{T}^{\mathbf{P}}$ ($\mathcal{T}^{\mathbf{p}}$) is empty. (The expressions $R_i(x)$ and r_j denote finite conjunctions of atoms $\bigwedge\limits_l Q_{il}(x)$ and $\bigwedge\limits_l q_{jl}$, respectively.)

Let $\{P_1, \ldots, P_K, P_{K+1} \ldots, P_M\}$, $0 \le K \le M$, and $\{p_1, \ldots, p_k, p_{k+1} \ldots, p_m\}$, $0 \le k \le m$, be sets of all (monadic) predicate symbols and propositional symbols, respectively, occurring in $\mathcal{T}$. Let Δ be the set of all mappings from $\{1, \ldots, M\}$ to $\{0, 1\}$, and Θ be the set of all mappings from $\{1, \ldots, m\}$ to $\{0, 1\}$. An element $\delta \in \Delta$ ($\theta \in \Theta$) is represented by the sequence $[\delta(1), \ldots, \delta(M)] \in \{0, 1\}^M$ ($[\theta(1), \ldots, \theta(m)] \in \{0, 1\}^m$). Let us call elements of Δ and Θ predicate and propositional *colours*, respectively. Let Γ be a subset of Δ, and θ be an element of Θ, and ρ be a map from C to Γ . A triple (Γ, θ, ρ) is called a *colour scheme*, and ρ is called a *constant distribution*.

Note 1. The notion of the colour scheme came, of course, from the well known method of the decidability proof for the monadic class in classical first-order logic (see, for example, [BGG97]). In our case we construct quotient structures based only on the predicates and propositions which occur in the temporal part of the problem, because only these symbols are really responsible for the satisfiability of temporal constraints. Besides, we have to consider so-called constant distributions, because unlike the classical case we cannot eliminate constants replacing them by existentially bounded variables – the monodicity property would be lost.

For every colour scheme $\mathcal{C} = \langle \Gamma, \theta, \rho \rangle$ let us construct the formulae $\mathcal{F}_{\mathcal{C}}$, $\mathcal{A}_{\mathcal{C}}$, $\mathcal{B}_{\mathcal{C}}$ in the following way. In the beginning for every $\gamma \in \Gamma$ and for θ introduce the conjunctions:

$$F_\gamma(x) = \bigwedge_{\gamma(i)=1} P_i(x) \wedge \bigwedge_{\gamma(i)=0} \neg P_i(x), \qquad F_\theta = \bigwedge_{\theta(i)=1} p_i \wedge \bigwedge_{\theta(i)=0} \neg p_i,$$

$$A_\gamma(x) = \bigwedge_{\gamma(i)=1 \& i \le K} P_i(x), \qquad A_\theta = \bigwedge_{\theta(i)=1 \& i \le k} p_i,$$

$$B_\gamma(x) = \bigwedge_{\gamma(i)=1 \& i \le K} R_i(x), \qquad B_\theta = \bigwedge_{\theta(i)=1 \& i \le k} r_i.$$

Now $\mathcal{F}_\mathcal{C}$, $\mathcal{A}_\mathcal{C}$, $\mathcal{B}_\mathcal{C}$ are of the following forms

$$\mathcal{F}_\mathcal{C} = \bigwedge_{\gamma\in\Gamma} \exists x F_\gamma(x) \wedge F_\theta \wedge \bigwedge_{c\in C} F_{\rho(c)}(c) \wedge \forall x \bigvee_{\gamma\in\Gamma} F_\gamma(x),$$

$$\mathcal{A}_\mathcal{C} = \bigwedge_{\gamma\in\Gamma} \exists x A_\gamma(x) \wedge A_\theta \wedge \bigwedge_{c\in C} A_{\rho(c)}(c) \wedge \forall x \bigvee_{\gamma\in\Gamma} A_\gamma(x),$$

$$\mathcal{B}_\mathcal{C} = \bigwedge_{\gamma\in\Gamma} \exists x B_\gamma(x) \wedge B_\theta \wedge \bigwedge_{c\in C} B_{\rho(c)}(c) \wedge \forall x \bigvee_{\gamma\in\Gamma} B_\gamma(x).$$

We can consider the formula $\mathcal{F}_\mathcal{C}$ as a 'categorical' formula specification of a quotient structure given by a colour scheme. In turn, the formula $\mathcal{A}_\mathcal{C}$ represents the part of this specification which is 'responsible' just for 'transferring' temporal requirements from the current world (quotient structure) to its immediate successors. The clause $(\Box\forall)(\mathcal{A}_\mathcal{C} \Rightarrow \bigcirc\mathcal{B}_\mathcal{C})$ is then called a *merged step rule*. Note that if both sets $\{i \mid i \leq K, \gamma \in \Gamma, \gamma(i) = 1\}$ and $\{i \mid i \leq k, \theta(i) = 1\}$ are empty the rule $(\mathcal{A}_\mathcal{C} \Rightarrow \bigcirc\mathcal{B}_\mathcal{C})$ degenerates to $(\mathbf{true} \Rightarrow \bigcirc\mathbf{true})$.

Example 2. Let us return to the temporal problem obtained in the example 1. The temporal part produces the following set of step merged clauses

1. $(p_1 \wedge \forall x P_1(x)) \Rightarrow \bigcirc(p_2 \wedge \forall x(P_2(x) \wedge P_1(x)))$,
2. $(p_1 \wedge \exists x P_1(x)) \Rightarrow \bigcirc(p_2 \wedge \exists x(P_2(x) \wedge P_1(x)))$,
3. $(\forall x P_1(x)) \Rightarrow \bigcirc(\forall x(P_2(x) \wedge P_1(x)))$,
4. $(\exists x P_1(x)) \Rightarrow \bigcirc(\exists x(P_2(x) \wedge P_1(x)))$.

For this problem $K = M = 2, k = m = 1$. The problem does not contain any constants, and in this case the colour schemes are defined as pairs of the form (Γ, θ).

The first merged rule corresponds to the colour scheme $(\{[1, \_, \_]\}, [1, \_, \_])$ (the subformula $\exists x P_1(x) \wedge \forall x P_1(x)$ is reduced to $\forall x P_1(x)$). The second rule corresponds to $(\{[1, \_, \_], [0, \_, \_]\}, [1, \_, \_])$ (as usual the value of the empty conjunction $\bigwedge_{i\in\emptyset} P_i(x)$ is **true**). The third and the fourth rules correspond to $(\{[1, \_, \_]\}, [0, \_, \_])$ and $(\{[1, \_, \_], [0, \_, \_]\}, [0, \_, \_])$, respectively.

The set of merged step rules for a problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is denoted by $\mathbf{m}\mathcal{T}$.

4 Resolution Procedure for Monodic Induction Free Problems

A problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is called *induction free* if $\mathcal{T}$ does not contain eventuality rules. In this section a derivation system based on a *step resolution rule* is given which is complete for the induction free monodic fragment.

Definition 1 (step resolution rule). *Let $\mathbf{m}\mathcal{T}$ be the set of merged rules of a problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$, $(\mathcal{A} \Rightarrow \bigcirc\mathcal{B}) \in \mathbf{m}\mathcal{T}$. Then the step resolution inference rule w.r.t. $\mathcal{U}$ is the rule*

$$\frac{\mathcal{A} \Rightarrow \bigcirc\mathcal{B}}{\neg\mathcal{A}} \ (\bigcirc_{res})$$

with the side condition that the set $\mathcal{U} \cup \{\mathcal{B}\}$ is unsatisfiable. [1]

[1] The side condition provides the rule with the second (implicit) premise $\mathbf{true} \Rightarrow \bigcirc\neg\mathcal{B}$ giving this rule a usual resolution form.

Note 2. The test whether the side condition is satisfied does not involve temporal reasoning and can be given to any first-order proof search procedure.

By Step$(\mathcal{U}, \mathbf{m}\mathcal{T})$ we denote the set of all formulae which are obtained by the step resolution rule w.r.t $\mathcal{U}$ from a merged clause $\mathcal{A} \Rightarrow \bigcirc\mathcal{B}$ in $\mathbf{m}\mathcal{T}$. Since $\mathbf{m}\mathcal{T}$ is finite the set Step$(\mathcal{U}, \mathbf{m}\mathcal{T})$ is also finite.

Lemma 1 (soundness of step resolution). *Let $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ be a temporal problem, and $\neg\mathcal{A} \in$ Step$(\mathcal{U}, \mathbf{m}\mathcal{T})$. Then $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is satisfiable if, and only if, $< \mathcal{U} \cup \{\neg\mathcal{A}\}, \mathcal{S}, \mathcal{T} >$ is satisfiable.*

We describe a proof procedure for $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ by a binary relation $\rhd$ on (universal) sets of formulae, which we call a *transition* or *derivation relation*. In this section we define the derivation relations by the condition that each step $\mathcal{U}_i \rhd \mathcal{U}_{i+1}$ consists of adding to the set $\mathcal{U}_i$ (to the *state* $\mathcal{U}_i$) a formula from Step$(\mathcal{U}_i, \mathbf{m}\mathcal{T})$. (In the next section this relation will be extended by a new *sometime resolution rule*). A finite sequence $\mathcal{U}_0 \rhd \mathcal{U}_1 \rhd \mathcal{U}_2 \rhd \ldots \rhd \mathcal{U}_n$, where $\mathcal{U}_0 = \mathcal{U}$, is called a *(theorem proving) derivation* for $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$.[2]

Definition 2 (termination rule and fair derivation). *A theorem proving derivation $\mathcal{U} = \mathcal{U}_0 \rhd \mathcal{U}_1 \rhd \ldots \rhd \mathcal{U}_n$, $n \geq 0$, for a problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is successfully terminated if the set $\mathcal{U}_n \cup \mathcal{S}$ is unsatisfiable. The theorem proving derivation for a problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is called* fair *if it either successfully terminates or, for any $i \geq 0$ and a formula $\neg\mathcal{A} \in$ Step$(\mathcal{U}_i, \mathbf{m}\mathcal{T})$, there is $j \geq i$ such that $\neg\mathcal{A} \in \mathcal{U}_j$.*

Note 3. We intentionally do not include in our consideration the classical concept of redundancy (see [BG01]) and deletion rules over sets of first-order formulae $\mathcal{U}_i$ because the main purpose of this paper is just new developments within temporal reasoning.

As we can see only the universal part is modified during the derivation, the temporal and initial parts of the problem remain unchanged.

Following [FDP01] we base our proof of completeness on a *behavior graph* for the problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$. Since, in this section, we are interested only in induction free problems we consider only so-called *eventuality free* behaviour graphs.

Definition 3 (eventuality free behaviour graph). *Given a problem $\mathcal{P} =< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ we construct a finite directed graph G as follows. Every node of G is a one-tuple $(\mathcal{C})$ where $\mathcal{C}$ is a colour scheme for $\mathcal{T}$ such that the set $\mathcal{U} \cup \mathcal{F}_\mathcal{C}$ is satisfiable.*

For each node $(\mathcal{C})$, $\mathcal{C} = (\Gamma, \theta, \rho)$, we construct an edge in G to a node $(\mathcal{C}')$, $\mathcal{C}' = (\Gamma', \theta', \rho')$, if $\mathcal{U} \wedge \mathcal{F}_{\mathcal{C}'} \wedge \mathcal{B}_\mathcal{C}$ is satisfiable. They are the only edges originating from $(\mathcal{C})$.

A node $(\mathcal{C})$ is designated as an initial node of G if $\mathcal{S} \wedge \mathcal{U} \wedge \mathcal{F}_\mathcal{C}$ is satisfiable.

The eventuality free behaviour graph *H of $\mathcal{P}$ is the full subgraph of G given by the set of nodes reachable from the initial nodes.*

[2] In reality we can keep the states $\mathcal{U}_i$ in the form which is the most suitable for applying a first-order theorem prover procedure. For example, for a classical resolution-based procedure they could be saturated sets of clauses [BG01].

It is easy to see that there is the following relation between behaviour graphs of two temporal problems when one of them is obtained by extending the universal part of another one.

Lemma 2. *Let $\mathcal{P}_1 =< \mathcal{U}_1, \mathcal{S}, \mathcal{T} >$ and $\mathcal{P}_2 =< \mathcal{U}_2, \mathcal{S}, \mathcal{T} >$ be two $\mathcal{TL}$ problems such that $\mathcal{U}_1 \subseteq \mathcal{U}_2$. Then the behaviour graph H_2 of $\mathcal{P}_2$ is a subgraph of the behaviour graph H_1 of $\mathcal{P}_1$.*

Proof The graph H_2 is the full subgraph of H_1 given by the set of nodes whose interpretations satisfy $\mathcal{U}_2$ and which are reachable from the initial nodes of H_1 whose interpretations also satisfy $\mathcal{U}_2$. ❑

In the remainder of this section we will refer to an eventuality free behaviour graph simply as a behaviour graph.

Definition 4 (suitable pairs). *Let $(\mathcal{C}, \mathcal{C}')$ where $\mathcal{C} = (\Gamma, \theta, \rho)$, $\mathcal{C}' = (\Gamma', \theta', \rho')$ be an (ordered) pair of colour schemes for $\mathcal{T}$. An ordered pair of predicate colours (γ, γ') where $\gamma \in \Gamma$, $\gamma' \in \Gamma'$ is called* suitable *if the formula $\mathcal{U} \wedge F_{\gamma'}(x) \wedge B_{\gamma}(x)$ is satisfiable. Similarly, the ordered pair of propositional colours (θ, θ') is suitable if $\mathcal{U} \wedge F_{\theta'} \wedge B_{\theta}$ is satisfiable. The ordered pair of constant distributions (ρ, ρ') is called suitable if, for every $c \in C$, the pair $(\rho(c), \rho'(c))$ is suitable.*

Lemma 3. *Let H be the behaviour graph of a problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ with an edge from a node $(\mathcal{C})$ to a node $(\mathcal{C}')$ of H, where $\mathcal{C} = (\Gamma, \theta, \rho)$ and $\mathcal{C}' = (\Gamma', \theta', \rho')$. Then*

- *for every $\gamma \in \Gamma$ there exists $\gamma' \in \Gamma'$ such that the pair (γ, γ') is suitable;*
- *for every $\gamma' \in \Gamma'$ there exists $\gamma \in \Gamma$ such that the pair (γ, γ') is suitable;*
- *the pair of propositional colours (θ, θ') is suitable;*
- *the pair of constant distributions (ρ, ρ') is suitable.*

Proof To prove the first item it is enough to note that satisfiability of the expression $\mathcal{U} \wedge \mathcal{F}_{\mathcal{C}'} \wedge \mathcal{B}_{\mathcal{C}}$ implies satisfiability of $\mathcal{U} \wedge (\forall x \bigvee\limits_{\gamma' \in \Gamma'} F_{\gamma'}(x)) \wedge \exists x B_{\gamma}(x)$. This, in turn, implies satisfiability of its logical consequence $\mathcal{U} \wedge \bigvee\limits_{\gamma' \in \Gamma'} \exists x (F_{\gamma'}(x) \wedge B_{\gamma}(x))$. So, one of the members of this disjunction must be satisfiable. The second item follows from the satisfiability of the formula $\mathcal{U} \wedge (\forall x \bigvee\limits_{\gamma \in \Gamma} B_{\gamma}(x)) \wedge \exists x F_{\gamma'}(x)$. Other items are proved similarly. ❑

Let H be the behaviour graph of a problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ and $\Pi = (\mathcal{C}_0), \ldots, (\mathcal{C}_n), \ldots$ be a path in H where $\mathcal{C}_i = (\Gamma_i, \theta_i, \rho_i)$. Let $\mathcal{G}_0 = \mathcal{S} \cup \{\mathcal{F}_{\mathcal{C}_o}\}$ and $\mathcal{G}_n = \mathcal{F}_{\mathcal{C}_n} \wedge \mathcal{B}_{\mathcal{C}_{n-1}}$ for $n \geq 1$. From classical model theory, since the language $\mathcal{L}$ is countable and does not contain equality the following lemma holds.

Lemma 4. *Let κ be a cardinal, $\kappa \geq \aleph_0$. For every $n \geq 0$, if a set $\mathcal{U} \cup \{\mathcal{G}_n\}$ is satisfiable, then there exists an $\mathcal{L}$-model $\mathfrak{M}_n = \langle D, I_n \rangle$ of $\mathcal{U} \cup \{\mathcal{G}_n\}$ such that for every $\gamma \in \Gamma_n$ the set $D_{(n,\gamma)} = \{a \in D \mid \mathfrak{M}_n \models F_{\gamma}(a)\}$ is of cardinality κ.*

Definition 5 (run). *By a* run *in Π we mean a function from $\mathbb{N}$ to $\bigcup_{i\in\mathbb{N}} \Gamma_i$ such that for every $n \in \mathbb{N}$, $r(n) \in \Gamma_n$ and the pair $(r(n), r(n+1))$ is suitable.*

It follows from the definition of H that for every $c \in C$ the function r_c defined by $r_c(n) = \rho_n(c)$ is a run in Π.

Theorem 2. *An* induction free *problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is satisfiable if, and only if, there exists an infinite path $\Pi = (\mathcal{C}_0), \ldots, (\mathcal{C}_n), \ldots$ through the behaviour graph H for $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ where $(\mathcal{C}_0)$ is an initial node of H.*

Proof ($\Rightarrow$) Let $\mathfrak{M} = \langle D, \mathcal{I}\rangle$ be a model of $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$. Let us define for every $n \in \mathbb{N}$ the node $(\mathcal{C})$, $\mathcal{C} = (\Gamma_n, \theta_n, \rho_n)$, as follows.

For every $a \in D$ let $\gamma_{(n,a)}$ be a map from $\{1, \ldots, M\}$ to $\{0, 1\}$, and let θ_n be a map from $\{1, \ldots, M\}$ to $\{0, 1\}$ such that

$$\gamma_{(n,a)}(i) \rightleftharpoons \begin{cases} 1, & \text{if } \mathfrak{M}_n \models P_i(a), \\ 0, & \text{if } \mathfrak{M}_n \not\models P_i(a) \end{cases} \qquad \theta_n(i) \rightleftharpoons \begin{cases} 1, & \text{if } \mathfrak{M}_n \models p_i, \\ 0, & \text{if } \mathfrak{M}_n \not\models p_i \end{cases}$$

for every $1 \leq i \leq M$.

Now we define $\Gamma_n \rightleftharpoons \{\gamma_{(n,a)} \mid a \in D\}$, and $\rho_n(c) \rightleftharpoons \gamma_{(n,c^{\mathcal{I}(n)})}$ for every $c \in C$. (Recall that, in accordance with our semantics, all constants are "rigid", that is $c^{I(u)} = c^{I(v)}$ for every $u, v \in \mathbb{N}$.) According to the construction $(\Gamma_n, \theta_n, \rho_n)$ given above we can conclude thet the sequence $(\mathcal{C}_0), \ldots, (\mathcal{C}_n), \ldots$ where $\mathcal{C}_n = (\Gamma_n, \theta_n, \rho_n)$, $n \in \mathbb{N}$, is a path through H.

Proof ($\Leftarrow$) Following [HWZ00] take a cardinal $\kappa \geq \aleph_0$ exceeding the cardinality of the set $\Re$ of all runs in Π. Let us define a domain $D = \{\langle r, \xi\rangle \mid r \in \Re, \xi < \kappa\}$. Then for every $n \in \mathbb{N}$ and for every $\delta \in \Delta$

$$\|\{\langle r, \xi\rangle \in D \mid r(n) = \delta\}\| = \begin{cases} \kappa, & \text{if } \delta \in \Gamma_n, \\ 0, & \text{otherwise.} \end{cases}$$

So, for every $n \in \mathbb{N}$ it follows that $D = \bigcup_{\gamma\in\Gamma_n} D_{(n,\gamma)}$ where $D_{(n,\gamma)} = \{\langle r, \xi\rangle \in D \mid r(n) = \gamma\}$. Hence by Lemma 4, for every $n \in \mathbb{N}$ there exists an $\mathcal{L}$-structure $\mathfrak{M}_n = \langle D, I_n\rangle$ which satisfies $\mathcal{U} \cup \{\mathcal{G}_n\}$. Moreover, we can suppose that $c^{I_n} = \langle r_c, 0\rangle$ and $D_{(n,\gamma)} = \{\langle r, \xi\rangle \in D \mid \mathfrak{M}_n \models F_\gamma(\langle r, \xi\rangle)\}$ for every $\gamma \in \Gamma_n$. A first-order temporal model that we sought is $\mathfrak{M} = \langle D, \mathcal{I}\rangle$ where $\mathcal{I}(n) = I_n$ for all $n \in \mathbb{N}$. To be convinced of that let us show validity of an arbitrary step rule $\Box(P_i(x) \Rightarrow \bigcirc R_i(x))$ in $\mathfrak{M}$. Namely, let us show that, for every $n \geq 0$ and for every $\langle r, \xi\rangle \in D$, if $\mathfrak{M}_n \models P_i(\langle r, \xi\rangle)$, then $\mathfrak{M}_{n+1} \models R_i(\langle r, \xi\rangle)$. Suppose $r(n) = \gamma \in \Gamma_n$ and $r(n+1) = \gamma' \in \Gamma_{n+1}$, that is $\langle r, \xi\rangle \in D_{(n,\gamma)}$ and $\langle r, \xi\rangle \in D_{(n+1,\gamma')}$. If $\mathfrak{M}_n \models P_i(\langle r, \xi\rangle)$ then $\gamma(i) = 1$. It follows that $R_i(x)$ is embedded in $B_\gamma(x)$ (if we consider $R_i(x)$ and $F_{\gamma'}(x)$ as sets). Since the pair (γ, γ') is suitable it follows that the conjunction $R_i(x)$ is embedded in $F_{\gamma'}(x)$. Together with $\mathfrak{M}_{n+1} \models F_{\gamma'}(\langle r, \xi\rangle)$ this implies that $\mathfrak{M}_{n+1} \models R_i(\langle r, \xi\rangle)$. ❑

Corollary 1 (completeness of the step resolution). *If an* induction free *problem $\mathcal{P} = < \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is unsatisfiable, then every* fair *theorem proving derivation for $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ succesfully terminates.*

Proof Let $\mathcal{U} = \mathcal{U}_0 \rhd \ldots \rhd \mathcal{U}_i \rhd \ldots \rhd \mathcal{U}_n$ be a fair theorem proving derivation for a problem $\mathcal{P} = < \mathcal{U}, \mathcal{S}, \mathcal{T} >$. The proof proceeds by induction on the number of nodes in the behaviour graph H of $\mathcal{P}$, which is finite. If H is empty then the set $\mathcal{U} \cup \mathcal{S}$ is unsatisfiable. In this case the derivation is successfully terminated because the set $\mathcal{U}_n \cup \mathcal{S}$ includes $\mathcal{U} \cup \mathcal{S}$ and therefore it is unsatisfiable too.

Now suppose H is not empty. Since $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is unsatisfiable Theorem 2 tells us that all paths through H starting from initial nodes are finite. Let $(\mathcal{C})$ be a node of H which has no successors. In this case the set $\mathcal{U} \cup \{\mathcal{B}_\mathcal{C}\}$ is unsatisfiable. Indeed, suppose $\mathcal{U} \cup \{\mathcal{B}_\mathcal{C}\}$ is satisfiable, and $\langle D', I' \rangle$ is a model of $\mathcal{U} \cup \{\mathcal{B}_\mathcal{C}\}$. Then following the proof of the previous theorem we can define a colour scheme $\mathcal{C}'$ such that $\langle D', I' \rangle \models \mathcal{F}_{\mathcal{C}'}$. Since $\mathcal{B}_\mathcal{C} \wedge \mathcal{F}_{\mathcal{C}'}$ is satisfiable there is an edge from the node $(\mathcal{C})$ to the node $\mathcal{C}'$ in the contradiction with the choice of $(\mathcal{C})$ as having no successor. Since the derivation is fair, there is a step when $\neg\mathcal{A}_\mathcal{C}$ is included to a state $\mathcal{U}_i \supseteq \mathcal{U}$. This implies removing the node $(\mathcal{C})$ from the behaviour graph H_i of the problem $< \mathcal{U}_i, \mathcal{S}, \mathcal{T} >$ because the set $\{\mathcal{F}_\mathcal{C}, \neg\mathcal{A}_\mathcal{C}\}$ is not satisfiable. By lemma 2 it follows that H_i is a proper subgraph of H.

Now we can apply induction hypothesis to the problem $< \mathcal{U}_i, \mathcal{S}, \mathcal{T} >$ and to the fair derivation $\mathcal{U}_i \rhd \ldots \rhd \mathcal{U}_n$. ❑

Example 3. Let us return to Example 2. We can apply step resolution (w.r.t. $\mathcal{U}_0$) to the second clause because the set $\mathcal{U}_0 \cup \{p_2 \wedge \forall x(P_2(x) \wedge P_1(x))\}$ is unsatisfiable:

$$\frac{(p_1 \wedge \forall x P_1(x)) \Rightarrow \bigcirc(p_2 \wedge \forall x(P_2(x) \wedge P_1(x)))}{\neg(p_1 \wedge \forall x P_1(x))} \ (\bigcirc_{res})$$

5 Resolution Procedure for Ground Eventuality Monodic Problems

A problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is called a *ground eventuality* problem if $\mathcal{T}$ contains only propositional eventuality rules. In this section a derivation system based on the step resolution rule defined above and on a new *sometime resolution rule* defined below is given which is complete for the ground eventuality monodic fragment.

Definition 6 (sometime resolution rule). *Let $\mathbf{m}\mathcal{T}$ be the set of merged rules of a problem $< \mathcal{U}, \mathcal{S}, \mathcal{T} >$, $\{\mathcal{A}_1 \Rightarrow \bigcirc\mathcal{B}_1, \ldots \mathcal{A}_n \Rightarrow \bigcirc\mathcal{B}_n\}$ is a subset of $\mathbf{m}\mathcal{T}$, and $p \Rightarrow \bigcirc\Diamond q$ is a propositional eventuality rule in $\mathcal{T}$. Then the sometime resolution inference rule w.r.t. $\mathcal{U}$ is the rule*

$$\frac{\mathcal{A}_1 \Rightarrow \bigcirc\mathcal{B}_1, \ldots, \mathcal{A}_n \Rightarrow \bigcirc\mathcal{B}_n \qquad p \Rightarrow \bigcirc\Diamond q}{\neg(\bigvee_{i=1}^{n} \mathcal{A}_i) \vee \neg p} \ (\Diamond_{res})$$

where the following (loop) side condition *has to be satisfied*

$$\mathcal{U} \cup \{\mathcal{B}_m\} \vdash \neg q \wedge \bigvee_{i=1}^{n} \mathcal{A}_i \quad \textit{for all } 1 \leq m \leq n.$$

Under the side condition given above $\bigvee_{i=1}^{n} \mathcal{A}_i$ can be considered as an *invariant formula* that provides the derivability of $\Box\bigcirc\neg q$ from $\mathcal{U} \cup \mathcal{T}$. Again, as in the case of step resolution, the test of whether the side conditions are satisfied does not involve temporal reasoning and can be given to any first-order proof search procedure.

By $\mathrm{Res}(\mathcal{U}, \mathcal{T})$ we denote the set of all formulae which are obtained by the sometime resolution rule w.r.t $\mathcal{U}$ from a set of merged clauses $\mathcal{A}_1 \Rightarrow \bigcirc\mathcal{B}_1, \ldots \mathcal{A}_n \Rightarrow \bigcirc\mathcal{B}_n$ in $\mathbf{m}\mathcal{T}$ and an eventuality clause $p \Rightarrow \bigcirc\Diamond q$ in $\mathcal{T}$. Since $\mathcal{T}$ and $\mathbf{m}\mathcal{T}$ are finite the set $\mathrm{Step}(\mathcal{U}, \mathcal{T})$ is also finite (up to renaming bound variables). The sometime resolution rule is sound in the sense similar to the soundness of the step resolution rule (see Lemma 1).

To take into account eventuality clauses we modify the notion of the behaviour graph given in the previous section by introducing an additional (eventuality) component to every node.

Definition 7 (ground eventuality behaviour graph).

Given a problem $\mathcal{P} =< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ we construct a finite directed graph G as follows. Every node of G is a two-tuple $(\mathcal{C}, E)$ where

- *$\mathcal{C}$ is a colour scheme for $\mathcal{T}$ such that the set $\mathcal{U} \wedge \mathcal{F}_{\mathcal{C}}$ is satisfiable;*
- *E is a subset of eventuality atoms occurring in $\mathcal{T}$. It will be called the* eventuality set *of the node $(\mathcal{C}, E)$.*

For each node $(\mathcal{C}, E)$, $\mathcal{C} = (\Gamma, \theta, \rho)$, we construct an edge in G to a node $(\mathcal{C}')$, $\mathcal{C}' = (\Gamma', \theta', \rho')$, if $\mathcal{U} \wedge \mathcal{F}_{\mathcal{C}'} \wedge \mathcal{B}_{\mathcal{C}}$ is satisfiable and $E' = E^1 \cup E^2$ where

$$E^1 = \{q \mid q \in E \text{ and } F_{\theta'} \not\vdash q\},$$
$$E^2 = \{q \mid \text{there exists an eventuality rule } (p \Rightarrow \bigcirc\Diamond q) \in \mathcal{T} \text{ such that } F_\theta \vdash p \text{ and } F_{\theta'} \not\vdash q\}.$$

They are the only edges originating from $(\mathcal{C}, E)$. A node $(\mathcal{C}, \emptyset)$ is designated as an initial node of G if $\mathcal{S} \wedge \mathcal{U} \wedge \mathcal{F}_{\mathcal{C}}$ is satisfiable. The eventuality free behaviour graph *H of $\mathcal{P}$ is the full subgraph of G given by the set of nodes reachable from the initial nodes.*

Let H be the behaviour graph of a problem $\mathcal{P}$, $\mathfrak{n}, \mathfrak{n}'$ be nodes of a graph H. We denote the relation "$\mathfrak{n}'$ is an immediate successor of $\mathfrak{n}$ " by $\mathfrak{n} \to \mathfrak{n}'$, and the relation "$\mathfrak{n}'$ is a successor of $\mathfrak{n}$ " by $\mathfrak{n} \to^+ \mathfrak{n}'$.

A node $\mathfrak{n}$ of H is called *step inference node* if it has no successors. A node $\mathfrak{n}'$ of H is called *sometime inference node* if it is not a step inference node and there is an eventuality atom q in $\mathcal{P}$ such that for every successor $\mathfrak{n}' = (\mathcal{C}', E')$, $q \in E'$ holds.

Lemma 5 (existence of a model).

Let $\mathcal{P}$ be a problem, H be the behaviour graph of $\mathcal{P}$ such that the set of initial nodes of H is not empty and the following condition is satisfied:

$$\forall\mathfrak{n}\forall q\exists\mathfrak{n}'(\mathfrak{n} \to^+ \mathfrak{n}' \wedge q \not\in E') \tag{1}$$

where $\mathfrak{n}, \mathfrak{n}'$ are nodes of H, $\mathfrak{n} = (\mathcal{C}, E)$, $\mathfrak{n}' = (\mathcal{C}', E')$, and q belongs to the set of eventuality atoms of $\mathcal{P}$. Then $\mathcal{P}$ has a model.

Proof We can construct a model for $\mathcal{P}$ as follows. Let $\mathfrak{n}_0$ be an initial node of H and $q_1, \ldots, q_m$ be all eventuality atoms of $\mathcal{P}$. Let Π be a path $\mathfrak{n}_0, \ldots, \mathfrak{n}_1, \ldots, \mathfrak{n}_m, \ldots, \mathfrak{n}_{m+1}, \ldots, \mathfrak{n}_{2m}, \ldots$, where $\mathfrak{n}_{km+j} = (\mathcal{C}_{km+j}, E_{km+j})$ is a successor of $\mathfrak{n}_{km+j-1}$ in H such that $q_j \not\in E_{km+j}$ (for every $k \geq 0$, $1 \leq j \leq m$).

Let us take the sequence $(\mathcal{C}_0), \ldots, (\mathcal{C}_1), \ldots, (\mathcal{C}_m), \ldots, (\mathcal{C}_{m+1}), \ldots, (\mathcal{C}_{2m}), \ldots$ induced by Π. Now let us consider this sequence as an infinite path in the eventuality free behaviour graph for the induction free problem $< \mathcal{U}, \mathcal{S}, \mathcal{T}^* >$ where $\mathcal{T}^*$ is obtained from $\mathcal{T}$ by removing all (propositional) eventuality rules.[3] Then the first-order temporal model $\mathfrak{M} = \langle D, \mathcal{I} \rangle$ constructed by the theorem 2 for $< \mathcal{U}, \mathcal{S}, \mathcal{T}^* >$ from the sequence $(\mathcal{C}_0), \ldots, (\mathcal{C}_1), \ldots, (\mathcal{C}_m), \ldots, (\mathcal{C}_{m+1}), \ldots, (\mathcal{C}_{2m}), \ldots$ is a model for $\mathcal{P} =< \mathcal{U}, \mathcal{S}, \mathcal{T} >$.

Indeed, all nontemporal clauses and all step clauses of $\mathcal{P}$ are satisfied on this structure immediately by the definition of $< \mathcal{U}, \mathcal{S}, \mathcal{T}^* >$. Let us take an arbitrary eventuality clause $p_j \Rightarrow \bigcirc \Diamond q_j$ of $\mathcal{T}$, a moment of time $l \in \mathbb{N}$ and the l-th element $(\mathcal{C}, E)$ on Π. If $F_{\mathcal{C}} \not\vdash p_j$ then $p_j \Rightarrow \bigcirc \Diamond q_j$ is satisfied at the moment l, i.e. $\mathfrak{M}_l \models (p_j \supset \bigcirc \Diamond q_j)$. If $F_{\mathcal{C}} \not\vdash p_j$ we take a node $\mathfrak{n}_{km+j}$ which is a successor of $(\mathcal{C}, E)$ on Π. By the construction of Π it follows that $q_j \not\in E_{km+j}$. We conclude that there exists a successor $(\mathcal{C}', E')$ of $(\mathcal{C}, E)$ along the path to $\mathfrak{n}_{km+j}$ such that $F_{\mathcal{C}'} \vdash q_j$, otherwise $l_j \in E_{km+j}$ would hold. It implies that $p_j \Rightarrow \bigcirc \Diamond q_j$ is satisfied at the moment l as well. ❑

To provide the completeness of the sometime resolution rule for the problems which contain more than one eventuality atom such problems have to be *augmented* in the following way.

Definition 8 (augmented problem). *Let us introduce for every eventuality atom q occurring in $\mathcal{T}$ a new propositional symbol w_q. An* augmented proplem $\mathcal{P}^{aug}$ *is a triple* $< \mathcal{U}^{aug}, \mathcal{S}, \mathcal{T}^{aug} >$ *where*

$$\mathcal{U}^{aug} = \mathcal{U} \cup \{w_q \supset (p \vee q) \mid (p \Rightarrow \bigcirc \Diamond q) \in \mathcal{T}\},$$
$$\mathcal{T}^{aug} = \mathcal{T} \cup \{p \Rightarrow \bigcirc w_q \mid (p \Rightarrow \bigcirc \Diamond q) \in \mathcal{T}\}.$$

The necessity for the augmentation even in the propositional case was shown in [DF00]. It is obvious that the augmentation is invariant with respect to satisfiability.[4]

Now we extend the notion of the derivation relation introduced in the previous section as follows: each step $\mathcal{U}_i \rhd \mathcal{U}_{i+1}$ consists of the adding to the set $\mathcal{U}_i$ a formula from $\mathrm{Step}(\mathcal{U}_i, \mathbf{m}\mathcal{T})$ or from $\mathrm{Res}(\mathcal{U}_i, \mathcal{T})$. Correspondingly, the notion of the (fair) theorem proving derivation is modified.

Theorem 3 (completeness of the step+sometime resolution). *If a ground eventuality problem $\mathcal{P} =< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is unsatisfiable, then every* fair *theorem proving derivation for $\mathcal{P}^{aug} =< \mathcal{U}^{aug}, \mathcal{S}, \mathcal{T}^{aug} >$ is succesfully terminated.*

[3] To retain the set of propositional symbols of $\mathcal{T}^*$ the same as of $\mathcal{T}$ we can add to $\mathcal{T}^*$ degenerates step rules of the form $p \Rightarrow \bigcirc \mathbf{true}$.

[4] Both the augmentation and including degenerates rules (see the previous footnote) can result in the violation of the condition that there are no different step rules with the same left-hand sides. However this violation is eliminated easy.

Proof The proof proceeds by induction on the number of nodes in the behaviour graph H of $\mathcal{P}^{aug}$, which is finite. The cases when H is empty graph or there exists a node $\mathfrak{n}$ in H which has no successors are considered in the same way as in the proof of Corollary 1.

Now we consider another possibility when H is not empty and every node in H has a successor. It is enough to prove that there is a formula $\psi \in \mathrm{Res}(\mathcal{U}, \mathcal{T})$ such that for some node $(\mathcal{C}, E)$ of H the formula $\mathcal{U} \wedge \mathcal{F}_{\mathcal{C}} \wedge \psi$ is unsatisfiable.

In this case because $\mathcal{P}$ is unsatisfiable the following condition (the negation of the condition (1) of the existence of a model given in Lemma 5) holds:

$$\exists \mathfrak{n} \exists q \forall \mathfrak{n}' (\mathfrak{n} \rightarrow^{+} \mathfrak{n}' \supset q \in E') \tag{2}$$

where $\mathfrak{n}, \mathfrak{n}'$ are nodes of H, $\mathfrak{n} = (I, E)$, $\mathfrak{n}' = (I', E')$, and q belongs to the set of eventuality atoms of $\mathcal{P}$.

Let $\mathfrak{n}_0 = (\mathcal{C}_0, E_0)$ be the node defined by the first existential quantifier of the condition (2). Let q_0 be the eventuality atom defined by the second existential quantifier of the condition (2). Let $p \Rightarrow \bigcirc \Diamond q_0$ be the eventuality rule containing q_0 (on the right).

Let $\mathfrak{I}$ be a finite nonempty set of indexes, $\{\mathfrak{n}_i \mid i \in \mathfrak{I}\}$ be the set of all successors of $\mathfrak{n}_0$. (It is possible, of course, that $0 \in \mathfrak{I}$.) Let $\mathfrak{n}_{i_1}, \ldots \mathfrak{n}_{i_k}$ be the set of all immediate successors of $\mathfrak{n}_0$, $\mathfrak{n}_{i_j} = (\mathcal{C}_{i_j}, E_{i_j})$ for $1 \leq j \leq k$. To simplify denotations in this proof we will represent merged rules $\mathcal{A}_{\mathcal{C}_i} \Rightarrow \bigcirc \mathcal{B}_{\mathcal{C}_i}$ ($\mathcal{A}_{\mathcal{C}_{i_j}} \Rightarrow \bigcirc \mathcal{B}_{\mathcal{C}_{i_j}}$) simply as $\mathcal{A}_i \Rightarrow \bigcirc \mathcal{B}_i$ ($\mathcal{A}_{i_j} \Rightarrow \bigcirc \mathcal{B}_{i_j}$), and formulae $\mathcal{F}_{\mathcal{C}_i}$ ($\mathcal{F}_{\mathcal{C}_{i_j}}$) simply as $\mathcal{F}_i$ ($\mathcal{F}_{i_j}$).

Consider two cases depending on whether the merged rule $\mathcal{A}_0 \Rightarrow \bigcirc \mathcal{B}_0$ (or any of $\mathcal{A}_i \Rightarrow \bigcirc \mathcal{B}_i$, $i \in \mathfrak{I}$) is degenerated or not.

1. Let $\mathcal{A}_0 = \mathcal{B}_0 = \mathbf{true}$. It implies, that $\mathcal{U} \vdash \neg q_0$. Indeed, since $q_0 \in E_{i_j}$ for all $1 \leq j \leq k$ then $\mathcal{F}_{i_j} \not\vdash q_0$ in accordance with the definition of the ground eventuality behaviour graph. Again similar to the proof of the Corollary 1, suppose that $\mathcal{U} \cup \{q_0\}$ is satisfiable, and $\langle D', I' \rangle$ is a model of $\mathcal{U} \cup \{q_0\}$. Then we can construct a colour scheme $\mathcal{C}'$ such that $\langle D', I' \rangle \models \mathcal{F}_{\mathcal{C}'}$ and therefore $\mathcal{F}_{\mathcal{C}'} \vdash q_0$. Since $\mathfrak{n}_{i_1}, \ldots \mathfrak{n}_{i_k}$ is the set of all immediate successors of $\mathfrak{n}_0$ and $\mathcal{B}_0 = \mathbf{true}$ it holds that there exists j, $1 \leq j \leq k$, such that $\mathcal{C}_{i_j} = \mathcal{C}'$. We conclude that $q_0 \notin E_{i_j}$ because of $\mathcal{F}_{\mathcal{C}'} \vdash q_0$. It contradicts the choice of the node $\mathfrak{n}_0$. So, $\mathcal{U} \vdash \neg q_0$, and the following sometime resolution inference is realized

$$\frac{\mathbf{true} \Rightarrow \bigcirc \mathbf{true} \qquad p \Rightarrow \bigcirc \Diamond q_0}{\neg p} \ (\Diamond_{res})$$

The behaviour graph for the problem $< \mathcal{U} \cup \{\neg p\}, \mathcal{S}, \mathcal{T} >$ is a proper subgraph of H. Indeed, if $F_{\theta_0} \vdash p$ then $\mathfrak{n}_0$ has to be removed from H. If $F_{\theta_0} \not\vdash p$ then a predecessor $(\mathcal{C}, E)$, $\mathcal{C} = (\Gamma, \theta, \rho)$, of the node $\mathfrak{n}_0$ such that $F_\theta \vdash p$ has to be removed from H. The set of such predecessors is not empty because the eventuality set of every initial node of H is empty.

The same argument holds if one of $\mathcal{A}_i \Rightarrow \bigcirc \mathcal{B}_i$, $i \in \mathfrak{I}$, is degenerate.

2. Let neither $\mathcal{A}_0 \Rightarrow \bigcirc \mathcal{B}_0$ nor any $\mathcal{A}_i \Rightarrow \bigcirc \mathcal{B}_i, i \in \mathfrak{I}$, are degenerate. We are going to prove now that in this case the sometime resolution rule

$$\frac{\{\mathcal{A}_i \Rightarrow \bigcirc \mathcal{B}_i \mid i \in \{0\} \cup \mathfrak{I}\} \qquad p \Rightarrow \bigcirc \Diamond q_0}{(\bigwedge_{i \in \{0\} \cup \mathfrak{I}} \neg \mathcal{A}_i) \vee \neg p} \ (\Diamond_{res})$$

is applied. We have to check the side conditions for the sometime resolution rule.

- By arguments similar those given in item 1 we conclude that the sets $(\mathcal{U} \cup \{\mathcal{B}_i\} \cup \{q_0\})$ for all $i \in \{0\} \cup \mathfrak{I}$ are unsatisfiable. It implies that $\mathcal{U} \cup \{\mathcal{B}_i\} \vdash \neg q_0$ for all $i \in \{0\} \cup \mathfrak{I}$.
- Let us show that $\mathcal{U} \cup \{\mathcal{B}_i\} \vdash \bigvee\limits_{j \in \{0\} \cup \mathfrak{I}} \mathcal{A}_j$ for all $i \in \{0\} \cup \mathfrak{I}$. Consider the case $i = 0$, for other indexes arguments are the same. Suppose that $(\mathcal{U} \cup \{\mathcal{B}_0\} \cup \{\bigwedge\limits_{1 \le j \le k} \neg \mathcal{A}_j\})$ is satisfied in a structure $\langle D', I' \rangle$. Let $\mathcal{C}'$ be a colour scheme of $\langle D', I' \rangle$, that is $\langle D', I' \rangle \models \mathcal{F}_{\mathcal{C}'}$. Then there is a node $\mathfrak{n}_{i_j} = (\mathcal{C}_{i_j}, E_{i_j})$, $1 \le j \le k$, which is an immediate successor of $\mathfrak{n}_0$, such that $\mathcal{C}_{i_j} = \mathcal{C}'$, and hence $\langle D', I' \rangle \models \mathcal{A}_{i_j}$. However it contradicts the choice of the structure $\langle D', I' \rangle$.

After applying the $(\Diamond_{res})$ rule given above we add to $\mathcal{U}$ its conclusion, which is equivalent to the set of formulae $\{\neg \mathcal{A}_i \vee \neg p \mid i \in \{0\} \cup \mathfrak{I}\}$. To prove that the behaviour graph of the extended problem will contain less nodes than H we have to consider two cases depending on whether $q_0 \in E_0$ or not.

a) Let us suppose $q_0 \notin E_0$. Then $\mathcal{F}_0 \vdash p$ because $q_0 \in E_{i_1}$ and there is an adge from $(\mathcal{C}_0, E_0)$ to $(\mathcal{C}_{i_1}, E_{i_1})$. In this case the node $\mathfrak{n}_0$ has to be removed from H. Recall that $\mathcal{F}_0 \vdash \mathcal{A}_0$ by the definition of $\mathcal{A}_0$.

b) Let us suppose $q_0 \in E_0$. Since the eventuality set of every initial node is empty there exists a predecessor $\mathfrak{n}'_0$ of $\mathfrak{n}_0$ and a path $\mathfrak{n}'_0 = (\mathcal{C}'_0, E'_0) \ldots, \mathfrak{n}'_m = (\mathcal{C}'_m, E'_m)$ from $\mathfrak{n}'_0$ to $\mathfrak{n}_0$, $m \ge 1$, $\mathcal{C}'_m = (\Gamma'_m, \theta'_m, \rho'_m)$, such that $\mathfrak{n}'_m = \mathfrak{n}_0$, $F_{\theta'_0} \vdash p$, and $q_0 \in E'_j$ for all $1 \le j \le m$. The last condition implies

$$F_{\theta'_j} \vdash \neg q_0 \text{ for all } 1 \le j \le m. \tag{3}$$

That is just the place where we have to involve in our arguments the augmenting pair for $p \Rightarrow \bigcirc \Diamond q_0$. Let this pair be presented by the following clauses

$$p \Rightarrow \bigcirc w_0 \quad \in \mathcal{T}^{aug}, \tag{4}$$

$$w_0 \Rightarrow q_0 \vee p \quad \in \mathcal{U}^{aug}. \tag{5}$$

From the clause (4) it follows that $F_{\theta'_1} \vdash w_0$. From the clause (5) and the condition (3) it follows that for all $1 \le j \le m$ it holds $F_{\theta'_j} \vdash p$. It implies that $F_{\theta_0} \vdash p$, in particular, since $\mathfrak{n}'_m = \mathfrak{n}_0$. So, $\mathcal{F}_{\mathcal{C}_0} \wedge \neg p$ is unsatisfiable. Therefore $\mathfrak{n}_o$ has to be removed from the behaviour graph after extending $\mathcal{U}^{aug}$ by the formula $\neg \mathcal{A}_0 \vee \neg p$ (the same as every node $\mathfrak{n}_i$, $I \in \mathfrak{I}$ is removed after including the formula $\neg \mathcal{A}_i \vee \neg p$.

Now all possible cases related to the properties of H have been considered. ❑

Lemma 6 (existence of a model, one eventuality case). *Let $\mathcal{P}$ be a problem such that $\mathcal{P}$ contains the only eventuality atom q_0. Let H be the behaviour graph of $\mathcal{P}$ such that the set of initial nodes of H is not empty and the following condition is satisfied:*

$$\forall \mathfrak{n} (q_0 \notin E \supset \exists \mathfrak{n}' (\mathfrak{n} \rightarrow^+ \mathfrak{n}' \wedge q_0 \notin E')) \tag{6}$$

where $\mathfrak{n}, \mathfrak{n}'$ are nodes of H, $\mathfrak{n} = (\mathcal{C}, E)$, $n' = (\mathcal{C}', E')$. Then $\mathcal{P}$ has a model.

Proof We use model construction of the proof of Lemma 5 taking $m = 1$. ❑

Corollary 2 (completeness of the one eventuality case). *If a ground eventuality problem $\mathcal{P} =< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is unsatisfiable, and $\mathcal{T}$ contains at most one eventuality atom then every* fair *theorem proving derivation for $\mathcal{P} =< \mathcal{U}, \mathcal{S}, \mathcal{T} >$ is succesfully terminated.*

Proof This corollary is obtained by analysing the proof of Theorem 3 given above. Firstly, using Lemma 6 and supposing q_0 to be the only eventuality atom of $\mathcal{T}$ we can strengthen the condition (2) to the following $\exists \mathfrak{n}(q_0 \not\in E \wedge \forall \mathfrak{n}'(\mathfrak{n} \rightarrow^+ \mathfrak{n}' \supset q_0 \in E'))$ where $\mathfrak{n}, \mathfrak{n}'$ are nodes of the eventuality graph H for the problem $\mathcal{P}$, $\mathfrak{n} = (\mathcal{C}, E)$, $\mathfrak{n}' = (\mathcal{C}', E')$. This immediately implies that the case 2(b) of the previous proof, where augmentation has been required, is excluded from the consideration. ❑

6 Conclusion

It has been known for a long time that first-order temporal logic over the natural numbers is incomplete [Sza86], that is there exists no finitary inference system which is sound and complete for the logic, or equivalently, the set of valid formulae of the logic is not recursively enumerable. The monodic fragment is the only known today fragment of first-order temporal logic among not only decidable but even recursively enumerable fragments which has a transparent syntactical definition and a finite inference system.

The method developed in this paper covers a special subsclass of the monodic fragment, namely the subclass of the *ground eventuality* monodic problems. Nevertheless this subclass is still interesting w.r.t. both its theoretical properties and possible area of applications. The first statement is confirmed in particular by the fact that if we slightly extend its boundaries admitting a binary relation in the step rules then its recursive enumerability will be lost. The second is justified in particular by the observation that the temporal specifications for verifying properties of transducers considered in [Spi00] are proved to be not simply monodic but monodic ground eventuality problems.

One of the essential advantages of the method given above follows from the complete separation of the classical first-order component. As a result classical first-order resolution can be applied as a basic tool in the temporal proof search (to solve side and termination conditions, which are expressed in classical first-order logic). That immediately gains access to all benefits, both theoretical and practical, of resolution based decision procedures [FLHT01], because the first-order formulae produced by temporal rules are very simple and they cannot change the decidability/undecidability of the initial fragment. Future work includes extending these results to wider fragments of first-order temporal logic, and implementing this approach.

It might also be interesting to decompose the present separated and 'global' temporal inferences into a mix of resolution-like 'local' rules. That will involve revision of the resolution method without skolemization for classical logic developed in [Zam87].

We thank anonymous referees for their helpful comments and suggestions. Unfortunately our possibilities to fully respond to them are limited by space restrictions. This work was supported by EPSRC under research grants GR/M46631 and GR/N08117.

References

[AVFY00] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proceedings of 17th Symposium on Database Systems (PODS'1998)*, pages 179–187. ACM Press, 2000.

[BG01] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 19–100. Elsevier Science and MIT Press, 2001.

[BGG97] E. Börger, E. Grädel, and Yu. Gurevich. *The Classical Decision Problem*. Springer Verlag, 1997.

[DF00] A. Degtyarev and M. Fisher. Propositional temporal resolution revised. In H.J. Ohlbach, editor, *Proc. of 7th UK Workshop on Automated Reasoning (ARW'00)*, London, U.K., June 2000.

[FDP01] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Transactions on Computation Logic*, 2(1), January 2001.

[Fis91] M. Fisher. A resolution method for temporal logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, 1991.

[Fis92] M. Fisher. A normal form for first-order temporal formulae. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, Saratoga Springs, USA, 1992. Springer Verlag.

[Fis97] M. Fisher. A normal form for temporal logics and its applications in theorem-proving and execution. *Journal of Logic and Computation*, 7(4), 1997.

[FLHT01] C.G. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1791–1852. Elsevier Science and MIT Press, 2001.

[Gab87] D. Gabbay. Declarative past and imperative future:executive temporal logic for interactive systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings on Colloquium on Temporal Logic and Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 402–450, Altrincham, U.K., 1987. Springer Verlag.

[HWZ00] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Fragments of first-order temporal logics. *Annals of Pure and Applied logic*, 106:85–134, 2000.

[PG86] D.A. Plaisted and S.A. Greenbaum. A structure-preserving clause form transformation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.

[Spi00] M. Spielmann. Verification of relational transducers for electronic commerce. In *Proceedings of 19th Symposium on Database Systems (PODS'2000)*, pages 92–103, Dallas, Texas, 2000. ACM Press.

[Sza86] A. Szalas. Concerning the semantic consequence relation in first-order temporal logic. *Theoretical Computer Science*, 47:329–334, 1986.

[Wol82] P. Wolper. *Syshthesis of Communicating Processes from Temporal Logic Specifications*. Ph.d. dissertation, Stanford University, 1982.

[Wol00] M. Wolter, F. and. ZAkharyaschev. Spatio-temporal representation and reasoning based on RCC-8. In *Proceedings of the 7th Conference on Principles of Knowledge Representation and Reasoning (KR'2000)*, pages 3–14, Montreal, Canada, 2000.

[WZ01] F. Wolter and M. Zakharyaschev. Axiomatizing the monodic fragment of first-order temporal logic. To appear in *Annals of Pure and Applied logic.*, 2001.

[Zam87] N. Zamov. The resolution method without skolemization. *Soviet Mathematical Doklady*, 293(5):1046–1049, 1987.

Approximating Most Specific Concepts in Description Logics with Existential Restrictions

Ralf Küsters[1] and Ralf Molitor[2⋆]

[1] Institut für Informatik und
Praktische Mathematik
Christian-Albrechts-Universität zu Kiel, Germany
kuesters@ti.informatik.uni-kiel.de
[2] IT Research & Development Group
Swiss Life
Zürich, Switzerland
ralf.molitor@swisslife.ch

Abstract. Computing the most specific concept (msc) is an inference task that allows to abstract from individuals defined in description logic (DL) knowledge bases. For DLs that allow for existential restrictions or number restrictions, however, the msc need not exist unless one allows for cyclic concepts interpreted with the greatest fixed-point semantics. Since such concepts cannot be handled by current DL-systems, we propose to approximate the msc. We show that for the DL $\mathcal{ALE}$, which has concept conjunction, a restricted form of negation, existential restrictions, and value restrictions as constructors, approximations of the msc always exist and can effectively be computed.

1 Introduction

The most specific concept (msc) of an individual b is a concept description that has b as instance and is the least concept description (w.r.t. subsumption) with this property. Roughly speaking, the msc is the concept description that, among all concept descriptions of a given DL, represents b best. Closely related to the msc is the least common subsumer (lcs), which, given concept descriptions $C_1, \ldots, C_n$, is the least concept description (w.r.t. subsumption) subsuming $C_1, \ldots, C_n$. Thus, where the msc generalizes an individual, the lcs generalizes a set of concept descriptions.

In [2,3,4], the msc (first introduced in [15]) and the lcs (first introduced in [5]) have been proposed to support the bottom-up construction of a knowledge base. The motivation comes from an application in chemical process engineering [17], where the process engineers construct the knowledge base (which consists of descriptions of standard building blocks of process models) as follows: First, they introduce several "typical" examples of a standard building block as individuals, and then they generalize (the descriptions of) these individuals into a concept

⋆ This work was carried out while the author was still at the LuFG Theoretische Informatik, RWTH Aachen, Germany.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 33–47, 2001.

description that (i) has all the individuals as instances, and (ii) is the most specific description satisfying property (i). The task of computing a concept description satisfying (i) and (ii) can be split into two subtasks: computing the msc of a single individual, and computing the lcs of a given finite number of concepts. The resulting concept description is then presented to the knowledge engineer, who can trim the description according to his needs before adding it to the knowledge base.

The lcs has been thoroughly investigated for (sublanguages of) Classic [5,2, 12,11], for DLs allowing for existential restrictions like $\mathcal{ALE}$ [3], and most recently, for $\mathcal{ALEN}$, a DL allowing for both existential and number restrictions [13]. For all these DLs, except for Classic in case attributes are interpreted as total functions [12], it has turned out that the lcs always exists and that it can effectively be computed. Prototypical implementations show that the lcs algorithms behave quite well in practice [7,4].

For the msc, the situation is not that rosy. For DLs allowing for number restrictions or existential restrictions, the msc does not exist in general. Hence, the first step in the bottom-up construction, namely computing the msc, cannot be performed. In [2], it has been shown that for $\mathcal{ALN}$, a sublanguage of Classic, the existence of the msc can be guaranteed if one allows for cyclic concept descriptions, i.e., concepts with cyclic definitions, interpreted by the greatest fixed-point semantics. Most likely, such concept descriptions would also guarantee the existence of the msc in DLs with existential restrictions. However, current DL-systems, like FaCT [10] and RACE [9], do not support this kind of cyclic concept descriptions: although they allow for cyclic definitions of concepts, these systems do not employ the greatest fixed-point semantics, but descriptive semantics. Consequently, cyclic concept descriptions returned by algorithms computing the msc cannot be processed by these systems.

In this paper, we therefore propose to approximate the msc. Roughly speaking, for some given non-negative integer k, the *k-approximation* of the msc of an individual b is the least concept description (w.r.t. subsumption) among all concept descriptions with b as instance and role depth at most k. That is, the set of potential most specific concepts is restricted to the set of concept descriptions with role depth bounded by k. For (sublanguages of) $\mathcal{ALE}$ we show that k-approximations always exist and that they can effectively be computed. Thus, when replacing "msc" by "k-approximation", the first step of the bottom-up construction can always be carried out. Although the original outcome of this step is only approximated, this might in fact suffice as a first suggestion to the knowledge engineer.

While for full $\mathcal{ALE}$ our k-approximation algorithm is of questionable practical use (since it employs a simple enumeration argument), we propose improved algorithms for the sublanguages $\mathcal{EL}$ and $\mathcal{EL}_\neg$ of $\mathcal{ALE}$. ($\mathcal{EL}$ allows for conjunction and existential restrictions, and $\mathcal{EL}_\neg$ additionally allows for a restricted form of negation.) Our approach for computing k-approximations in these sublanguages is based on representing concept descriptions by certain trees and ABoxes by certain (systems of) graphs, and then characterizing instance relationships by homomorphisms from trees into graphs. The k-approximation operation then

Table 1. Syntax and semantics of concept descriptions.

Construct name	Syntax	Semantics	$\mathcal{EL}$	$\mathcal{EL}_\neg$	$\mathcal{ALE}$
top-concept	$\top$	Δ	x	x	x
bottom-concept	$\bot$	$\emptyset$		x	x
conjunction	$C \sqcap D$	$C^I \cap D^I$	x	x	x
existential restriction	$\exists r.C$	$\{x \in \Delta \mid \exists y : (x,y) \in r^I \wedge y \in C^I\}$	x	x	x
primitive negation	$\neg P$	$\Delta \setminus P^I$		x	x
value restriction	$\forall r.C$	$\{x \in \Delta \mid \forall y : (x,y) \in r^I \rightarrow y \in C^I\}$			x

consists in unraveling the graphs into trees and translating them back into concept descriptions. In case the unraveling yields finite trees, the corresponding concept descriptions are "exact" most specific concepts, showing that in this case the msc exists. Otherwise, pruning the infinite trees on level k yields k-approximations of the most specific concepts.

The outline of the paper is as follows. In the next section, we introduce the basic notions and formally define k-approximations. To get started, in Section 3 we present the characterization of instance relationships in $\mathcal{EL}$ and show how this can be employed to compute k-approximations or (if it exists) the msc. In the subsequent section we extend the results to $\mathcal{EL}_\neg$, and finally deal with $\mathcal{ALE}$ in Section 5. The paper concludes with some remarks on future work. Due to space limitations, we refer to [14] for all technical details and complete proofs.

2 Preliminaries

Concept descriptions are inductively defined with the help of a set of *constructors*, starting with disjoint sets N_C of *concept names* and N_R of *role names*. In this work, we consider concept descriptions built from the constructors shown in Table 1, where $r \in N_R$ denotes a role name, $P \in N_C$ a concept name, and C, D concept descriptions. The concept descriptions in the DLs $\mathcal{EL}$, $\mathcal{EL}_\neg$, and $\mathcal{ALE}$ are built using certain subsets of these constructors, as shown in the last three columns of Table 1.

An *ABox* $\mathcal{A}$ is a finite set of assertions of the form $(a,b) : r$ (*role assertions*) or $a : C$ (*concept assertions*), where a, b are *individuals* from a set N_I (disjoint from $N_C \cup N_R$), r is a role name, and C is a concept description. An ABox is called $\mathcal{L}$-ABox if all concept descriptions occurring in $\mathcal{A}$ are $\mathcal{L}$-concept descriptions.

The semantics of a concept description is defined in terms of an *interpretation* $\mathcal{I} = (\Delta, \cdot^\mathcal{I})$. The domain Δ of $\mathcal{I}$ is a non-empty set of objects and the interpretation function $\cdot^\mathcal{I}$ maps each concept name $P \in N_C$ to a set $P^I \subseteq \Delta$, each role name $r \in N_R$ to a binary relation $r^I \subseteq \Delta \times \Delta$, and each individual $b \in N_I$ to an element $a^\mathcal{I} \in \Delta$ such that $a \neq b$ implies $a^\mathcal{I} \neq b^\mathcal{I}$ (*unique name assumption*). The extension of $\cdot^\mathcal{I}$ to arbitrary concept descriptions is inductively defined, as shown in the third column of Table 1. An interpretation $\mathcal{I}$ is a *model*

of an ABox $\mathcal{A}$ iff it satisfies $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ for all role assertions $(a, b) : r \in \mathcal{A}$, and $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all concept assertions $a : C \in \mathcal{A}$.

The most important traditional inference services provided by DL-systems are computing the subsumption hierarchy and instance relationships. The concept description C is *subsumed* by the concept description D ($C \sqsubseteq D$) iff $C^I \subseteq D^I$ for all interpretations $\mathcal{I}$; C and D are *equivalent* ($C \equiv D$) iff they subsume each other. An individual $a \in N_I$ is an *instance* of C w.r.t. $\mathcal{A}$ ($a \in_{\mathcal{A}} C$) iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all models $\mathcal{I}$ of $\mathcal{A}$.

In this paper, we are interested in the computation of most specific concepts and their approximation via concept descriptions of limited depth. The *depth* $\mathsf{depth}(C)$ of a concept description C is defined as the maximum of nested quantifiers in C. We also need to introduce least common subsumers formally.

Definition 1 (msc, k-approximation, lcs). *Let $\mathcal{A}$ be an $\mathcal{L}$-ABox, a an individual in $\mathcal{A}$, $C, C_1, \ldots, C_n$ $\mathcal{L}$-concept descriptions, and $k \in \mathbb{N}$. Then,*

- *C is the* most specific concept *(msc) of a w.r.t. $\mathcal{A}$ ($\mathsf{msc}_{\mathcal{A}}(a)$) iff $a \in_{\mathcal{A}} C$, and for all $\mathcal{L}$-concept descriptions C', $a \in_{\mathcal{A}} C'$ implies $C \sqsubseteq C'$;*
- *C is the* k-approximation *of a w.r.t. $\mathcal{A}$ ($\mathsf{msc}_{k,\mathcal{A}}(a)$) iff $a \in_{\mathcal{A}} C$, $\mathsf{depth}(C) \leq k$, and for all $\mathcal{L}$-concept descriptions C', $a \in_{\mathcal{A}} C'$ and $\mathsf{depth}(C') \leq k$ imply $C \sqsubseteq C'$;*
- *C is the* least common subsumer *(lcs) of $C_1, \ldots, C_n$ ($\mathsf{lcs}(C_1, \ldots, C_n)$) iff $C_i \sqsubseteq C$ for all $i = 1, \ldots, n$, and for all $\mathcal{L}$-concept descriptions C', $C_i \sqsubseteq C'$ for all $i = 1, \ldots, n$ implies $C \sqsubseteq C'$.*

Note that by definition, most specific concepts, k-approximations, and least common subsumers are uniquely determined up to equivalence (if they exist).

The following example shows that in DLs allowing for existential restrictions the msc of an ABox-individual b need not exist.

Example 1. Let $\mathcal{L}$ be one of the DLs $\mathcal{EL}$, $\mathcal{EL}_\neg$, or $\mathcal{ALE}$. Consider the $\mathcal{L}$-ABox $\mathcal{A} = \{(b, b) : r\}$. It is easy to see that, for each $n \geq 0$, b is an instance of the $\mathcal{L}$-concept description

$$C_n := \underbrace{\exists r. \cdots \exists r}_{n \text{ times}}. \top.$$

The msc of b can be written as the infinite conjunction $\bigsqcap_{n \geq 0} C_n$, which, however, cannot be expressed by a (finite) $\mathcal{L}$-concept description. As we will see, the k-approximation of b is the $\mathcal{L}$-concept description $\bigsqcap_{0 \leq n \leq k} C_n$.

3 Most Specific Concepts in $\mathcal{EL}$

In the following subsection, we introduce the characterization of instance relationships in $\mathcal{EL}$, which yields the basis for the algorithm computing k-approximations (Section 3.2). All results presented in this section are rather straightforward. However, they prepare the ground for the more involved technical problems one encounters for $\mathcal{EL}_\neg$.

3.1 Characterizing Instance in $\mathcal{EL}$

In order to characterize instance relationships, we need to introduce description graphs (representing ABoxes) and description trees (representing concept descriptions). For $\mathcal{EL}$, an *$\mathcal{EL}$-description graph* is a labeled graph of the form $\mathcal{G} = (V, E, \ell)$ whose edges $vrw \in E$ are labeled with role names $r \in N_R$ and whose nodes $v \in V$ are labeled with sets $\ell(v)$ of concept names from N_C. The empty label corresponds to the top-concept. An *$\mathcal{EL}$-description tree* is of the form $\mathcal{G} = (V, E, v_0, \ell)$, where (V, E, ℓ) is an $\mathcal{EL}$-description graph which is a tree with root $v_0 \in V$.

$\mathcal{EL}$-concept descriptions can be turned into $\mathcal{EL}$-description trees and vice versa [3]: Every $\mathcal{EL}$-concept description C can be written (modulo equivalence) as $C \equiv P_1 \sqcap \ldots \sqcap P_n \sqcap \exists r_1.C_1 \sqcap \ldots \sqcap \exists r_m.C_m$ with $P_i \in N_C \cup \{\top\}$. Such a concept description is recursively translated into an $\mathcal{EL}$-description tree $\mathcal{G}(C) = (V, E, v_0, \ell)$ as follows: The set of all concept names P_i occurring on the top-level of C yields the label $\ell(v_0)$ of the root v_0, and each existential restriction $\exists r_i.C_i$ yields an r_i-successor that is the root of the tree corresponding to C_i. For example, the concept description

$$C := \exists s.(Q \sqcap \exists r.\top) \sqcap \exists r.(Q \sqcap \exists s.\top)$$

yields the description tree depicted on the left hand side of Figure 1.

Every $\mathcal{EL}$-description tree $\mathcal{G} = (V, E, v_0, \ell)$ is translated into an $\mathcal{EL}$-concept description $C_\mathcal{G}$ as follows: the concept names occurring in the label of v_0 yield the concept names in the top-level conjunction of $C_\mathcal{G}$, and each $v_0 r v \in E$ yields an existential restriction $\exists r.C$, where C is the $\mathcal{EL}$-concept description obtained by translating the subtree of $\mathcal{G}$ with root v. For a leaf $v \in V$, the empty label is translated into the top-concept.

Adapting the translation of $\mathcal{EL}$-concept descriptions into $\mathcal{EL}$-description trees, an $\mathcal{EL}$-ABox $\mathcal{A}$ is translated into an $\mathcal{EL}$-description graph $\mathcal{G}(\mathcal{A})$ as follows: Let $\mathsf{Ind}(\mathcal{A})$ denote the set of all individuals occurring in $\mathcal{A}$. For each $a \in \mathsf{Ind}(\mathcal{A})$, let $C_a := \bigsqcap_{a:D \in \mathcal{A}} D$, if there exists a concept assertion $a : D \in \mathcal{A}$, otherwise, $C_a := \top$. Let $\mathcal{G}(C_a) = (V_a, E_a, a, \ell_a)$ denote the $\mathcal{EL}$-description tree obtained from C_a. (Note that the individual a is defined to be the root of $\mathcal{G}(C_a)$; in particular, $a \in V_a$.) W.l.o.g. let the sets V_a, $a \in \mathsf{Ind}(\mathcal{A})$ be pairwise disjoint. Then, $\mathcal{G}(\mathcal{A}) := (V, E, \ell)$ is defined by

- $V := \bigcup_{a \in \mathsf{Ind}(\mathcal{A})} V_a$,
- $E := \{arb \mid (a, b) : r \in \mathcal{A}\} \cup \bigcup_{a \in \mathsf{Ind}(\mathcal{A})} E_a$, and
- $\ell(v) := \ell_a(v)$ for $v \in V_a$.

As an example, consider the $\mathcal{EL}$-ABox

$$\begin{aligned}\mathcal{A} = \{&a : P \sqcap \exists s.(Q \sqcap \exists r.P \sqcap \exists s.\top), b : P \sqcap Q, c : \exists r.P,\\ &(a, b) : r, (a, c) : r, (b, c) : s\}.\end{aligned}$$

The corresponding $\mathcal{EL}$-description graph $\mathcal{G}(\mathcal{A})$ is depicted on the right hand side of Figure 1.

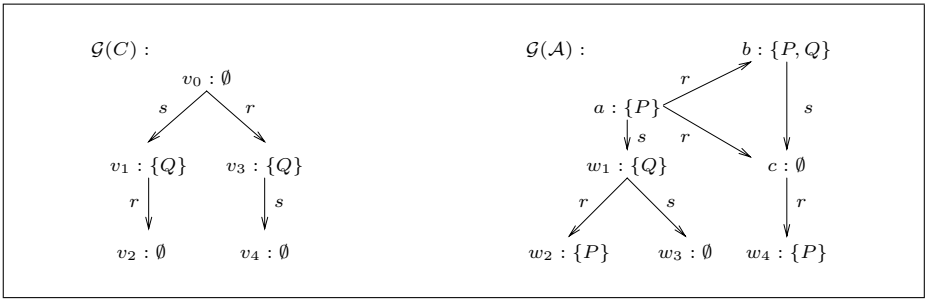


Fig. 1. The $\mathcal{EL}$-description tree of C and the $\mathcal{EL}$-description graph of $\mathcal{A}$.

Now, an instance relationship, $a \in_{\mathcal{A}} C$, in $\mathcal{EL}$ can be characterized in terms of a homomorphism from the description tree of C into the description graph of $\mathcal{A}$: a *homomorphism* from an $\mathcal{EL}$-description tree $\mathcal{G} = (V, E, v_0, \ell)$ into an $\mathcal{EL}$-description graph $\mathcal{H} = (V_H, E_H, \ell_H)$ is a mapping $\varphi : V \longrightarrow V_H$ such that (1) $\ell(v) \subseteq \ell_H(\varphi(v))$ for all $v \in V$, and (2) $\varphi(v)r\varphi(w) \in E_H$ for all $vrw \in E$.

Theorem 1. *[14] Let $\mathcal{A}$ be an $\mathcal{EL}$-ABox, $a \in \mathsf{Ind}(\mathcal{A})$ an individual in $\mathcal{A}$, and C an $\mathcal{EL}$-concept description. Let $\mathcal{G}(\mathcal{A})$ denote the $\mathcal{EL}$-description graph of $\mathcal{A}$ and $\mathcal{G}(C)$ the $\mathcal{EL}$-description tree of C. Then, $a \in_{\mathcal{A}} C$ iff there exists a homomorphism φ from $\mathcal{G}(C)$ into $\mathcal{G}(\mathcal{A})$ such that $\varphi(v_0) = a$, where v_0 is the root of $\mathcal{G}(C)$.*

In our example, a is an instance of C, since mapping v_0 on a, v_i on w_i, $i = 1, 2$, and v_3 on b and v_4 on c yields a homomorphism from $\mathcal{G}(C)$ into $\mathcal{G}(\mathcal{A})$.

Theorem 1 is a special case of the characterization of subsumption between simple conceptual graphs [6], and of the characterization of containment of conjunctive queries [1]. In these more general settings, testing for the existence of homomorphisms is an NP-complete problem. In the restricted case of testing homomorphisms mapping trees into graphs, the problem is polynomial [8]. Thus, as corollary of Theorem 1, we obtain the following complexity result.

Corollary 1. *The instance problem for $\mathcal{EL}$ can be decided in polynomial time.*

Theorem 1 generalizes the following characterization of subsumption in $\mathcal{EL}$ introduced in [3]. This characterization uses homomorphisms between description trees, which are defined just as homomorphisms from description trees into description graphs, but where we additionally require to map roots onto roots.

Theorem 2. *[3] Let C, D be $\mathcal{EL}$-concept descriptions, and let $\mathcal{G}(C)$ and $\mathcal{G}(D)$ be the corresponding description trees. Then, $C \sqsubseteq D$ iff there exists a homomorphism from $\mathcal{G}(D)$ into $\mathcal{G}(C)$.*

3.2 Computing k-Approximations in $\mathcal{EL}$

In the sequel, we assume $\mathcal{A}$ to be an $\mathcal{EL}$-ABox, a an individual occurring in $\mathcal{A}$, and k a non-negative integer. Roughly, our algorithm computing $\mathsf{msc}_{k,\mathcal{A}}(a)$

works as follows: First, the description graph $\mathcal{G}(\mathcal{A})$ is unraveled into a tree with root a. This tree, denoted $\mathcal{T}(a, \mathcal{G}(\mathcal{A}))$, has a finite branching factor, but possibly infinitely long paths. Pruning all paths to length k yields an $\mathcal{EL}$-description tree $\mathcal{T}_k(a, \mathcal{G}(\mathcal{A}))$ of depth $\leq k$. Using Theorem 1 and Theorem 2, one can show that the $\mathcal{EL}$-concept description $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$ is equivalent to $\mathsf{msc}_{k,\mathcal{A}}(a)$. As an immediate consequence, in case $\mathcal{T}(a, \mathcal{G}(\mathcal{A}))$ is finite, $C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A}))}$ yields the msc of a. In what follows, we define $\mathcal{T}(a, \mathcal{G}(\mathcal{A}))$ and $\mathcal{T}_k(a, \mathcal{G}(\mathcal{A}))$, and prove correctness of our algorithm.

First, we need some notations: For an $\mathcal{EL}$-description graph $\mathcal{G} = (V, E, \ell)$, $p = v_0 r_1 v_1 r_2 \dots r_n v_n$ is a *path from* v_0 *to* v_n of *length* $|p| = n$ and with *label* $r_1 \cdots r_n$, if $v_{i-1} r_i v_i \in E$ for all $1 \leq i \leq n$. The empty path ($n = 0$) is allowed, in which case the label of p is the empty word ε. The node v_n is an $r_1 \cdots r_n$-*successor of* v_0. Every node is an ε-successor of itself. A node v is *reachable* from v_0, if there exists a path from v_0 to v. The graph $\mathcal{G}$ is *cyclic*, if there exists a non-empty path from a node in $\mathcal{G}$ to itself.

Definition 2. *Let* $\mathcal{G} = (V, E, \ell)$ *and* $a \in V$. *The* tree $\mathcal{T}(a, \mathcal{G})$ of a w.r.t. $\mathcal{A}$ *is defined by* $\mathcal{T}(a, \mathcal{G}) := (V^T, E^T, a, \ell^T)$ *with*

- $V^T := \{p \mid p$ *is a path from* a *to some node in* $\mathcal{G}\}$,
- $E^T := \{prq \mid p, q \in V^T$ *and* $q = prw$ *for some* $r \in N_R$ *and* $w \in V\}$,
- $\ell^T(p) := \ell(v)$ *if* p *is a path to* v.

For $k \in \mathbf{N}$, *the* tree $\mathcal{T}_k(a, \mathcal{G})$ of a w.r.t. $\mathcal{G}$ and k *is defined by* $\mathcal{T}_k(a, \mathcal{G}) := (V_k^T, E_k^T, a, \ell_k^T)$ *with*

- $V_k^T := \{p \in V^T \mid |p| \leq k\}$,
- $E_k^T := E^T \cap (V_k^T \times N_R \times V_k^T)$, *and*
- $\ell_k^T(p) := \ell^T(p)$ *for* $p \in V_k^T$.

Now, we can show the main theorem of this section.

Theorem 3. *Let* $\mathcal{A}$ *be an* $\mathcal{EL}$*-ABox,* $a \in \mathsf{Ind}(\mathcal{A})$, *and* $k \in \mathbf{N}$. *Then,* $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$ *is the* k*-approximation of* a *w.r.t.* $\mathcal{A}$. *If, starting from* a, *no cyclic path in* $\mathcal{A}$ *can be reached (i.e.,* $\mathcal{T}(a, \mathcal{G}(\mathcal{A}))$ *is finite), then* $C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A}))}$ *is the msc of* a *w.r.t.* $\mathcal{A}$; *otherwise no msc exists.*

Proof sketch. Obviously, there exists a homomorphism from $\mathcal{T}_k(a, \mathcal{G}(\mathcal{A}))$, a tree isomorphic to $\mathcal{G}(C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))})$, into $\mathcal{G}(\mathcal{A})$ with a mapped on a. By Theorem 1, this implies $a \in_{\mathcal{A}} C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$.

Let C be an $\mathcal{EL}$-concept description with $a \in_{\mathcal{A}} C$ and $\mathsf{depth}(C) \leq k$. Theorem 1 implies that there exists a homomorphism φ from $\mathcal{G}(C)$ into $\mathcal{G}(\mathcal{A})$. Given φ, it is easy to construct a homomorphism from $\mathcal{G}(C)$ into $\mathcal{T}_k(a, \mathcal{G}(\mathcal{A}))$. Thus, with Theorem 2, we conclude $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))} \sqsubseteq C$. Altogether, this shows that $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$ is a k-approximation as claimed.

Now, assume that, starting from a, a cycle can be reached in $\mathcal{A}$, that is, $\mathcal{T}(a, \mathcal{G}(\mathcal{A}))$ is infinite. Then, we have a decreasing chain $C_0 \sqsupset C_1 \sqsupset \cdots$ of k-approximations C_k ($\equiv C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$) with increasing depth k, $k \geq 0$. From Theorem 2, we conclude that there does not exist an $\mathcal{EL}$-concept description

subsumed by all of these k-approximations (since such a concept description only has a fixed and finite depth). Thus, a cannot have an msc.

Conversely, if $\mathcal{T}(a, \mathcal{G}(\mathcal{A}))$ is finite, say with depth k, from the observation that all k'-approximations, for $k' \geq k$, are equivalent, it immediately follows that $C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A}))}$ is the msc of a. □

Obviously, there exists a deterministic algorithm computing the k-approximation (i.e., $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$) in time $\mathcal{O}(|\mathcal{A}|^k)$. The *size* $|\mathcal{A}|$ of $\mathcal{A}$ is defined by

$$|\mathcal{A}| := |\mathsf{Ind}(\mathcal{A})| + |\{(a, b) : r \mid (a, b) : r \in \mathcal{A}\}| + \sum_{a:C \in \mathcal{A}} |C|,$$

where the size $|C|$ of C is defined as the sum of the number of occurrences of concept names, role names, and constructors in C. Similarly, one obtains an exponential complexity upper bound for computing the msc (if it exists).

Corollary 2. *For an $\mathcal{EL}$-ABox $\mathcal{A}$, an individual $a \in \mathsf{Ind}(\mathcal{A})$, and $k \in \mathbb{N}$, the k-approximation of a w.r.t. $\mathcal{A}$ always exists and can be computed in time $\mathcal{O}(|\mathcal{A}|^k)$.*

The msc of a exists iff starting from a no cycle can be reached in $\mathcal{A}$. The existence of the msc can be decided in polynomial time, and if the msc exists, it can be computed in time exponential in the size of $\mathcal{A}$.

In the remainder of this section, we prove that the exponential upper bounds are tight. To this end, we show examples demonstrating that k-approximations and the msc may grow exponentially.

Example 2. Let $\mathcal{A} = \{(a, a) : r, (a, a) : s\}$. The $\mathcal{EL}$-description graph $\mathcal{G}(\mathcal{A})$ as well as the $\mathcal{EL}$-description trees $\mathcal{T}_1(a, \mathcal{G}(\mathcal{A}))$ and $\mathcal{T}_2(a, \mathcal{G}(\mathcal{A}))$ are depicted in Figure 2. It is easy to see that, for $k \geq 1$, $\mathcal{T}_k(a, \mathcal{G}(\mathcal{A}))$ yields a full binary tree of depth k where

- each node is labeled with the empty set, and
- each node except the leaves has one r- and one s-successor.

By Theorem 3, $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$ is the k-approximation of a w.r.t. $\mathcal{A}$. The size of $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$ is $|\mathcal{A}|^k$. Moreover, it is not hard to see that there does not exist an $\mathcal{EL}$-concept description C which is equivalent to but smaller than $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$.

The following example illustrates that, if it exists, also the msc can be of exponential size.

Example 3. For $n \geq 1$, define $\mathcal{A}_n := \{(a_i, a_{i+1}) : r, (a_i, a_{i+1}) : s \mid 1 \leq i < n\}$. Obviously, $\mathcal{A}_n$ is acyclic, and the size of $\mathcal{A}_n$ is linear in n. By Theorem 3, $C_{\mathcal{T}(a_1,\mathcal{A}_n)}$ is the msc of a_1 w.r.t. $\mathcal{A}_n$. It is easy to see that, for each n, $\mathcal{T}(a_1, \mathcal{A}_n)$ coincides with the tree $\mathcal{T}_n(a, \mathcal{G}(\mathcal{A}))$ obtained in Example 2. As before we obtain that

- $C_{\mathcal{T}(a_1,\mathcal{G}(\mathcal{A}))}$ is of size exponential in $|\mathcal{A}_n|$; and
- there does not exist an $\mathcal{EL}$-concept description C equivalent to but smaller than $C_{\mathcal{T}(a_1,\mathcal{G}(\mathcal{A}))}$.

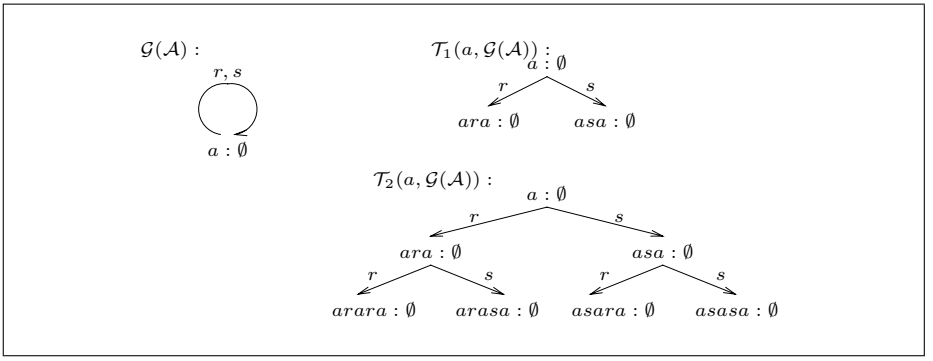


Fig. 2. The $\mathcal{EL}$-description graph and the $\mathcal{EL}$-description trees from Example 2.

Summarizing, we obtain the following lower bounds.

Proposition 1. *Let $\mathcal{A}$ be an $\mathcal{EL}$-ABox, $a \in \mathsf{Ind}(\mathcal{A})$, and $k \in \mathbf{N}$.*

- *The size of $\mathsf{msc}_{\mathcal{A},k}(a)$ may grow with $|\mathcal{A}|^k$.*
- *If it exists, the size of $\mathsf{msc}_{\mathcal{A}}(a)$ may grow exponentially in $|\mathcal{A}|$.*

4 Most Specific Concepts in $\mathcal{EL}_\neg$

Our goal is to obtain a characterization of the (k-approximation of the) msc in $\mathcal{EL}_\neg$ analogously to the one given in Theorem 3 for $\mathcal{EL}$. To achieve this goal, first the notions of description graph and of description tree are extended from $\mathcal{EL}$ to $\mathcal{EL}_\neg$ by allowing for subsets of $N_C \cup \{\neg P \mid P \in N_C\} \cup \{\bot\}$ as node labels. Just as for $\mathcal{EL}$, there exists a 1–1 correspondence between $\mathcal{EL}_\neg$-concept descriptions and $\mathcal{EL}_\neg$-description trees, and an $\mathcal{EL}_\neg$-ABox $\mathcal{A}$ is translated into an $\mathcal{EL}_\neg$-description graph $\mathcal{G}(\mathcal{A})$ as described for $\mathcal{EL}$-ABoxes. The notion of a homomorphism also remains unchanged for $\mathcal{EL}_\neg$, and the characterization of subsumption extends to $\mathcal{EL}_\neg$ by just considering inconsistent $\mathcal{EL}_\neg$-concept descriptions as a special case: $C \sqsubseteq D$ iff $C \equiv \bot$ or there exists a homomorphism φ from $\mathcal{G}(D)$ into $\mathcal{G}(C)$.

Second, we have to cope with inconsistent $\mathcal{EL}_\neg$-ABoxes as a special case: for an inconsistent ABox $\mathcal{A}$, $a \in_{\mathcal{A}} C$ is valid for all concept descriptions C, and hence, $\mathsf{msc}_{\mathcal{A}}(a) \equiv \bot$. However, extending Theorem 1 with this special case does not yield a sound and complete characterization of instance relationships for $\mathcal{EL}_\neg$. If this was the case, we would get that the instance problem for $\mathcal{EL}_\neg$ is in P, in contradiction to complexity results shown in [16], which imply that the instance problem for $\mathcal{EL}_\neg$ is coNP-hard.

The following example is an abstract version of an example given in [16]; it illustrates incompleteness of a naïve extension of Theorem 1 from $\mathcal{EL}$ to $\mathcal{EL}_\neg$.

Example 4. Consider the $\mathcal{EL}_\neg$-concept description $C = P \sqcap \exists r.(P \sqcap \exists r.\neg P)$ and the $\mathcal{EL}_\neg$-ABox $\mathcal{A} = \{a : P, b_1 : P, b_3 : \neg P, (a, b_1) : r, (a, b_2) : r, (b_1, b_2) : r, (b_2, b_3) : r\}$; $\mathcal{G}(\mathcal{A})$ and $\mathcal{G}(C)$ are depicted in Figure 3. Obviously, there does not exist

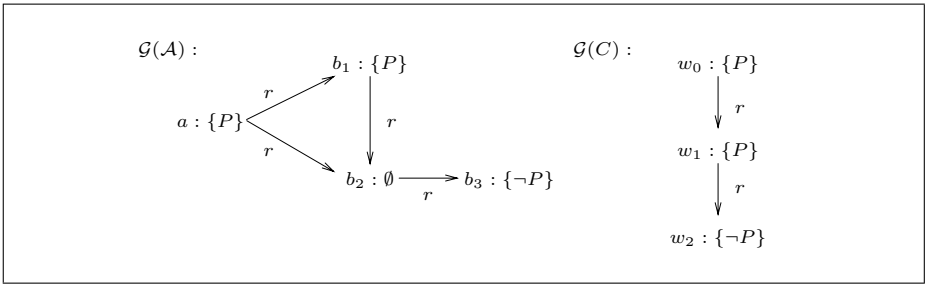


Fig. 3. The $\mathcal{EL}_\neg$-description graph and the $\mathcal{EL}_\neg$-description tree from Example 4.

a homomorphism φ from $\mathcal{G}(C)$ into $\mathcal{G}(\mathcal{A})$ with $\varphi(w_0) = a$, because neither $P \in \ell(b_2)$ nor $\neg P \in \ell(b_2)$. For each model $\mathcal{I}$ of $\mathcal{A}$, however, either $b_2^\mathcal{I} \in P^\mathcal{I}$ or $b_2^\mathcal{I} \in (\neg P)^\mathcal{I}$, and in fact, $a^\mathcal{I} \in C^\mathcal{I}$. Thus, a is an instance of C w.r.t. $\mathcal{A}$ though there does not exist a homomorphism φ from $\mathcal{G}(C)$ into $\mathcal{G}(\mathcal{A})$ with $\varphi(w_0) = a$.

In the following section, we give a sound and complete characterization of instance relationships in $\mathcal{EL}_\neg$, which again yields the basis for the characterization of k-approximations given in Section 4.2.

4.1 Characterizing Instance in $\mathcal{EL}_\neg$

The reason for the problem illustrated in Example 4 is that, in general, for the individuals in the ABox it is not always fixed whether they are instances of a given concept name or not. Thus, in order to obtain a sound and complete characterization analogous to Theorem 1, instead of $\mathcal{G}(\mathcal{A})$, one has to consider all so-called atomic completions of $\mathcal{G}(\mathcal{A})$.

Definition 3 (Atomic completion). *Let $\mathcal{G} = (V, E, \ell)$ be an $\mathcal{EL}_\neg$-description graph and $N_C^* := \{P \in N_C \mid$ exists $v \in V$ with $P \in \ell(v)$ or $\neg P \in \ell(v)\}$. An $\mathcal{EL}_\neg$-description graph $\mathcal{G}^* = (V, E, \ell^*)$ is an* atomic completion *of $\mathcal{G}$ if, for all $v \in V$,*

1. *$\ell(v) \subseteq \ell^*(v)$,*
2. *for all concept names $P \in N_C^*$ either $P \in \ell^*(v)$ or $\neg P \in \ell^*(v)$.*

Note that by definition, all labels of nodes in completions do not contain a conflict, i.e., the nodes are not labeled with a concept name and its negation. In particular, if $\mathcal{G}$ has a conflicting node, then $\mathcal{G}$ does not have a completion. It is easy to see that an $\mathcal{EL}_\neg$-ABox $\mathcal{A}$ is inconsistent iff $\mathcal{G}(\mathcal{A})$ contains a conflicting node. For this reason, in the following characterization of the instance relationship, we do not need to distinguish between consistent and inconsistent ABoxes.

Theorem 4. *[14] Let $\mathcal{A}$ be an $\mathcal{EL}_\neg$-ABox, $\mathcal{G}(\mathcal{A}) = (V, E, \ell)$ the corresponding description graph, C an $\mathcal{EL}_\neg$-concept description, $\mathcal{G}(C) = (V_C, E_C, w_0, \ell_C)$ the corresponding description tree, and $a \in \mathsf{Ind}(\mathcal{A})$. Then, $a \in_\mathcal{A} C$ iff for each atomic completion $\mathcal{G}(\mathcal{A})^*$ of $\mathcal{G}(\mathcal{A})$, there exists a homomorphism φ from $\mathcal{G}(C)$ into $\mathcal{G}(\mathcal{A})^*$ with $\varphi(w_0) = a$.*

The problem of deciding whether there exists an atomic completion $\mathcal{G}(\mathcal{A})^*$ such that there exists no homomorphism from $\mathcal{G}(C)$ into $\mathcal{G}(\mathcal{A})^*$ is in coNP. Adding the coNP-hardness result obtained from [16], this shows

Corollary 3. *The instance problem for $\mathcal{EL}_\neg$ is coNP-complete.*

4.2 Computing k-Approximations in $\mathcal{EL}_\neg$

Not surprisingly, the algorithm computing the k-approximation/msc in $\mathcal{EL}$ does not yield the desired result for $\mathcal{EL}_\neg$. For instance, in Example 4, we would get $C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A}))} = P \sqcap \exists r.\exists r.(\neg P) \sqcap \exists r.(P \sqcap \exists r.\exists r.(\neg P))$. But as we will see, $\mathsf{msc}_{\mathcal{A}}(a) \equiv P \sqcap \exists r.(P \sqcap \exists r.\neg P) \sqcap \exists r.(P \sqcap \exists r.\exists r.\neg P)$, i.e., $\mathsf{msc}_{\mathcal{A}}(a) \sqsubset C_{\mathcal{T}(a,\mathcal{A})}$.

As in the extension of the characterization of instance relationships from $\mathcal{EL}$ to $\mathcal{EL}_\neg$, we have to take into account all atomic completions instead of the single description graph $\mathcal{G}(\mathcal{A})$. Intuitively, one has to compute the least concept description for which there exists a homomorphism into each atomic completion of $\mathcal{G}(\mathcal{A})$. In fact, this can be done by applying the lcs operation on the set of all concept descriptions $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^*)}$ obtained from the atomic completions $\mathcal{G}(\mathcal{A})^*$ of $\mathcal{G}(\mathcal{A})$.

Theorem 5. *Let $\mathcal{A}$ be an $\mathcal{EL}_\neg$-ABox, $a \in \mathsf{Ind}(\mathcal{A})$, and $k \in \mathbf{N}$. If $\mathcal{A}$ is inconsistent, then $\mathsf{msc}_{k,\mathcal{A}}(a) \equiv \mathsf{msc}_{\mathcal{A}}(a) \equiv \bot$. Otherwise, let $\{\mathcal{G}(\mathcal{A})^1, \ldots, \mathcal{G}(\mathcal{A})^n\}$ be the set of all atomic completions of $\mathcal{G}(\mathcal{A})$.*

Then, $\mathsf{lcs}(C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^1)}, \ldots, C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^n)}) \equiv \mathsf{msc}_{k,\mathcal{A}}(a)$. If, starting from a, no cycle can be reached in $\mathcal{A}$, then $\mathsf{lcs}(C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A})^1)}, \ldots, C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A})^n)}) \equiv \mathsf{msc}_{\mathcal{A}}(a)$; otherwise the msc does not exist.

Proof sketch. Let $\mathcal{A}$ be a consistent $\mathcal{EL}_\neg$-ABox and $\mathcal{G}(\mathcal{A})^1, \ldots, \mathcal{G}(\mathcal{A})^n$ the atomic completions of $\mathcal{G}(\mathcal{A})$. By definition of $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^i)}$, there exists a homomorphism π_i from $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^i)}$ into $\mathcal{G}(\mathcal{A})^i$ for all $1 \leq i \leq n$. Let C_k denote the lcs of $\{C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^1)}, \ldots, C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^n)}\}$. The characterization of subsumption for $\mathcal{EL}_\neg$ yields homomorphisms φ_i from $\mathcal{G}(C_k)$ into $\mathcal{G}(C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^i)})$ for all $1 \leq i \leq n$. Now it is easy to see that $\pi_i \circ \varphi_i$ yields a homomorphism from $\mathcal{G}(C_k)$ into $\mathcal{G}(\mathcal{A})^i$, $1 \leq i \leq n$, each mapping the root of $\mathcal{G}(C_k)$ onto a. Hence, $a \in_{\mathcal{A}} C_k$.

Assume C' with $\mathsf{depth}(C') \leq k$ and $a \in_{\mathcal{A}} C'$. By Theorem 4, there exist homomorphisms ψ_i from $\mathcal{G}(C')$ into $\mathcal{G}(\mathcal{A})^i$ for all $1 \leq i \leq n$, each mapping the root of $\mathcal{G}(C')$ onto a. Since $\mathsf{depth}(C') \leq k$, these homomorphisms immediately yield homomorphisms ψ_i' from $\mathcal{G}(C')$ into $\mathcal{G}(C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^i)})$ for all $1 \leq i \leq n$. Now the characterization of subsumption yields $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A})^i)} \sqsubseteq C'$ for all $1 \leq i \leq n$, and hence $C_k \sqsubseteq C'$. Thus, $C_k \equiv \mathsf{msc}_{k,\mathcal{A}}(a)$.

Analogously, in case starting from a, no cycle can be reached in $\mathcal{A}$, we conclude $\mathsf{lcs}(C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A})^1)}, \ldots, C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A})^n)}) \equiv \mathsf{msc}_{\mathcal{A}}(a)$. Otherwise, with the same argument as in the proof of Theorem 3, it follows that the msc does not exist. □

In Example 4, we obtain two atomic completions, namely $\mathcal{G}(\mathcal{A})^1$ with $\ell^1(b_2) = \{P\}$, and $\mathcal{G}(\mathcal{A})^2$ with $\ell^2(b_2) = \{\neg P\}$. Now Theorem 5 implies $\mathsf{msc}_{\mathcal{A}}(a) \equiv \mathsf{lcs}(C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A})^1)}, C_{\mathcal{T}(a,\mathcal{G}(\mathcal{A})^2)})$, which is equivalent to

$$P \sqcap \exists r.(P \sqcap \exists r.\neg P) \sqcap \exists r.(P \sqcap \exists r.\exists r.\neg P).$$

The examples showing the exponential blow-up of the size of k-approximations and most specific concepts in $\mathcal{EL}$ can easily be adapted to $\mathcal{EL}_\neg$. However, we only have a double exponential upper bound (though we strongly conjecture that the size can again be bounded single-exponentially): the size of each tree (and the corresponding concept descriptions) obtained from an atomic completion is at most exponential, and the size of the lcs of a sequence of $\mathcal{EL}_\neg$-concept descriptions can grow exponentially in the size of the input descriptions [3].

Moreover, by an algorithm computing the lcs of the concept descriptions obtained from the atomic completions, the k-approximation (the msc) can be computed in double exponential time.

Corollary 4. *Let $\mathcal{A}$ be an $\mathcal{EL}_\neg$-ABox, $a \in \mathsf{Ind}(\mathcal{A})$, and $k \in \mathbb{N}$.*

- *The k-approximation of a always exists. It may be of size $|\mathcal{A}|^k$ and can be computed in double-exponential time.*
- *The msc of a exists iff $\mathcal{A}$ is inconsistent, or starting from a, no cycle can be reached in $\mathcal{A}$. If the msc exists, its size may grow exponentially in $|\mathcal{A}|$, and it can be computed in double-exponential time. The existence of the msc can be decided in polynomial time.*

5 Most Specific Concepts in $\mathcal{ALE}$

As already mentioned in the introduction, the characterization of instance relationships could not yet be extended from $\mathcal{EL}_\neg$ to $\mathcal{ALE}$. Since these structural characterizations were crucial for the algorithms computing the (k-approximation of the) msc in $\mathcal{EL}$ and $\mathcal{EL}_\neg$, no similar algorithms for $\mathcal{ALE}$ can be presented here. However, we show that

1. given that N_C and N_R are finite sets, the $\mathsf{msc}_{k,\mathcal{A}}(a)$ always exists and can effectively be computed (cf. Theorem 6);
2. the characterization of instance relationships in $\mathcal{EL}$ is also sound for $\mathcal{ALE}$ (cf. Lemma 1), which allows for approximating the k-approximation; and
3. we illustrate the main problems encountered in the structural characterization of instance relationships in $\mathcal{ALE}$ (cf. Example 5).

The first result is achieved by a rather generic argument. Given that the signature, i.e., the sets N_C and N_R, are fixed and finite, it is easy to see that also the set of $\mathcal{ALE}$-concept descriptions of depth $\leq k$ built using only names from $N_C \cup N_R$ is finite (up to equivalence) and can effectively be computed. Since the instance problem for $\mathcal{ALE}$ is known to be decidable [16], enumerating this set and retrieving the least concept description which has a as instance, obviously yields an algorithm computing $\mathsf{msc}_{k,\mathcal{A}}(a)$.

Theorem 6. *Let N_C and N_R be fixed and finite, and let $\mathcal{A}$ be an $\mathcal{ALE}$-ABox built over a set N_I of individuals and $N_C \cup N_R$. Then, for $k \in \mathbb{N}$ and $a \in \mathsf{Ind}(\mathcal{A})$, the k-approximation of a w.r.t. $\mathcal{A}$ always exists and can effectively be computed.*

Note that the above argument cannot be adapted to prove the existence of the msc for acyclic $\mathcal{ALE}$-ABoxes unless the size of the msc can be bounded appropriately. Finding such a bound remains an open problem.

The algorithm sketched above is obviously not applicable in real applications. Thus, in the remainder of this section, we focus on extending the improved algorithms obtained for $\mathcal{EL}$ and $\mathcal{EL}_\neg$ to $\mathcal{ALE}$.

5.1 Approximating the k-Approximation in $\mathcal{ALE}$

We first have to extend the notions of description graph and of description tree from $\mathcal{EL}_\neg$ to $\mathcal{ALE}$: In order to cope with value restrictions occurring in $\mathcal{ALE}$-concept descriptions, we allow for two types of edges, namely those labeled with role names $r \in N_R$ (representing existential restrictions of the form $\exists r.C$) and those labeled with $\forall r$ (representing value restrictions of the form $\forall r.C$). Again, there is a 1–1 correspondence between $\mathcal{ALE}$-concept descriptions and $\mathcal{ALE}$-description trees, and an $\mathcal{ALE}$-ABox $\mathcal{A}$ is translated into an $\mathcal{ALE}$-description graph $\mathcal{G}(\mathcal{A})$ just as described for $\mathcal{EL}$-ABoxes. The notion of a homomorphism also extends to $\mathcal{ALE}$ in a natural way. A homomorphism φ from an $\mathcal{ALE}$-description tree $\mathcal{H} = (V_H, E_H, v_0, \ell_H)$ into an $\mathcal{ALE}$-description graph $\mathcal{G} = (V, E, \ell)$ is a mapping $\varphi : V_H \longrightarrow V$ satisfying the conditions (1) and (2) on homomorphisms between $\mathcal{EL}$-description trees and $\mathcal{EL}$-description graphs, and additionally (3) $\varphi(v) \forall r \varphi(w) \in E$ for all $v \forall r w \in E_H$.

We are now ready to formalize soundness of the characterization of instance relationships for $\mathcal{ALE}$.

Lemma 1. *[14] Let $\mathcal{A}$ be an $\mathcal{ALE}$-ABox with $\mathcal{G}(\mathcal{A}) = (V, E, \ell)$, C an $\mathcal{ALE}$-concept description with $\mathcal{G}_C = (V_C, E_C, v_0, \ell_C)$, and $a \in \mathsf{Ind}(\mathcal{A})$.*

If there exists a homomorphism φ from $\mathcal{G}_C$ into $\mathcal{G}(\mathcal{A})$ with $\varphi(v_0) = a$, then $a \in_{\mathcal{A}} C$.

As an immediate consequence of this lemma, we get $a \in_{\mathcal{A}} C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$ for all $k \geq 0$, where the trees $\mathcal{T}(a, \mathcal{G}(\mathcal{A}))$ and $\mathcal{T}_k(a, \mathcal{G}(\mathcal{A}))$ are defined just as for $\mathcal{EL}$. This in turn yields $\mathsf{msc}_{k,\mathcal{A}}(a) \sqsubseteq C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))}$ and hence, an algorithm computing an approximation of the k-approximation for $\mathcal{ALE}$. In fact, such approximations already turned out to be quite usable in our process engineering application [4].

The following example now shows that the characterization is not complete for $\mathcal{ALE}$, and that, in general, $C_{\mathcal{T}_k(a,\mathcal{G}(\mathcal{A}))} \not\equiv \mathsf{msc}_{k,\mathcal{A}}(a)$. In particular, it demonstrates the difficulties one encounters in the presence of value restrictions.

Example 5. Consider the $\mathcal{ALE}$-ABox

$$\begin{aligned}\mathcal{A} := \{&a : P, b_1 : P \sqcap \forall s.P \sqcap \exists r.P, b_2 : P \sqcap \exists r.(P \sqcap \exists s : P),\\ &(a, b_1) : r, (a, b_2) : r, (b_1, b_2) : r\},\end{aligned}$$

and the $\mathcal{ALE}$-concept description $C = \exists r.(\forall s.P \sqcap \exists r.\exists s.\top)$; $\mathcal{G}(\mathcal{A})$ and $\mathcal{G}(C)$ are depicted in Figure 4. Note that $\mathcal{G}(\mathcal{A})$ is the unique atomic completion of itself (w.r.t. $N_C = \{P\}$).

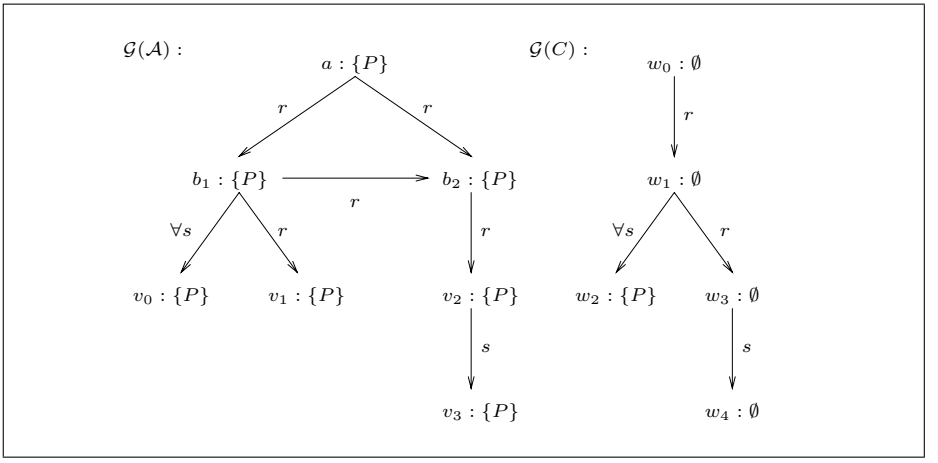


Fig. 4. The $\mathcal{ALE}$-description graph and the $\mathcal{ALE}$-description tree from Example 5.

It is easy to see that there does not exist a homomorphism φ from $\mathcal{G}(C)$ into $\mathcal{G}(\mathcal{A})$ with $\varphi(w_0) = a$. However, $a \in_{\mathcal{A}} C$: For each model $\mathcal{I}$ of $\mathcal{A}$, either $b_2^{\mathcal{I}}$ does not have an s-successor, or at least one s-successor. In the first case, $b_2^{\mathcal{I}} \in \forall s.P$, and hence $b_2^{\mathcal{I}}$ yields the desired r-successor of $a^{\mathcal{I}}$ in $(\forall s.P \sqcap \exists r.\exists s.\top)^{\mathcal{I}}$. In the second case, it is $b_2^{\mathcal{I}} \in (\exists s.\top)^{\mathcal{I}}$, and hence $b_1^{\mathcal{I}}$ yields the desired r-successor of $a^{\mathcal{I}}$. Thus, for each model $\mathcal{I}$ of $\mathcal{A}$, $a^{\mathcal{I}} \in C^{\mathcal{I}}$.

Moreover, for $k = 4$, $C_{\mathcal{T}_4(a,\mathcal{A})}$ is given by $P \sqcap \exists r.(P \sqcap \forall s.P \sqcap \exists r.P \sqcap \exists r.(P \sqcap \exists r.(P \sqcap \exists s.P))) \sqcap \exists r.(P \sqcap \exists r.(P \sqcap \exists s.P))$. It is easy to see that $C_{\mathcal{T}_4(a,\mathcal{A})} \not\sqsubseteq C$. Hence, $C_{\mathcal{T}_4(a,\mathcal{A})} \sqcap C \sqsubset C_{\mathcal{T}_4(a,\mathcal{A})}$, which implies $\mathsf{msc}_{4,\mathcal{A}}(a) \sqsubset C_{\mathcal{T}_4(a,\mathcal{A})}$.

Intuitively, the above example suggests that, in the definition of atomic completions, one should take into account not only (negated) concept names but also more complex concept descriptions. However, it is not clear whether an appropriate set of such concept descriptions can be obtained just from the ABox and how these concept descriptions need to be integrated in the completion in order to obtain a sound and complete structural characterization of instance relationships in $\mathcal{ALE}$.

6 Conclusion

Starting with the formal definition of the k-approximation of msc we showed that, for $\mathcal{ALE}$ and a finite signature (N_C, N_R), the k-approximation of the msc of an individual b always exists and can effectively be computed. For the sublanguages $\mathcal{EL}$ and $\mathcal{EL}_\neg$, we gave sound and complete characterizations of instance relationships that lead to practical algorithms. As a by-product, we obtained a characterization of the existence of the msc in $\mathcal{EL}$-/$\mathcal{EL}_\neg$-ABoxes, and showed that the msc can effectively be computed in case it exists.

First experiments with manually computed approximations of the msc in the process engineering application were quite encouraging [4]: used as inputs

for the lcs operation, i.e., the second step in the bottom-up construction of the knowledge base, they led to descriptions of building blocks the engineers could use to refine their knowledge base. In next steps, the run-time behavior and the quality of the output of the algorithms presented here is to be evaluated by a prototype implementation in the process engineering application.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. F. Baader and R. Küsters. Computing the least common subsumer and the most specific concept in the presence of cyclic $\mathcal{ALN}$-concept descriptions. In *Proc. of KI'98*, LNAI 1504, pp 129-140. Springer-Verlag, 1998.
3. F. Baader, R. Küsters, and R. Molitor. Computing least common subsumers in description logics with existential restrictions. In *Proc. of IJCAI'99*, pp 96-101. Morgan Kaufmann, 1999.
4. F. Baader and R. Molitor. Building and structuring description logic knowledge bases using least common subsumers and concept analysis. In *Proc. of ICCS2000*, LNAI 1867,pp 292-305. Springer-Verlag, 2000.
5. W.W. Cohen, A. Borgida, and H. Hirsh. Computing least common subsumers in description logics. In *Proc. of AAAI'92*, pp 754-760. MIT Press, 1992.
6. M. Chein and M. Mugnier. Conceptual graphs: Fundamental notions. *Revue d'Intelligence*. 6(4):365-406, 1992.
7. W.W. Cohen and H. Hirsh. Learning the CLASSIC description logic: Theoretical and experimental results. In *Proc. of KR'94*, pp 121-132. Morgan Kaufmann, 1994.
8. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
9. V. Haarslev and R. Möller. Expressive ABox reasoning with number restrictions, role hierarchies and transitively closed roles. In *Proc. of KR2000*, pp 273-284. Morgan Kaufmann, 2000.
10. I. Horrocks. Using an expressive description Logic: FaCT or fiction? In *Proc. of KR'98*, pp 636-647. Morgan Kaufmann, 1998.
11. R. Küsters. *Non-Standard Inference in Description Logics*. PhD thesis, RWTH Aachen, 2000. To appear as volume 2100 of the Springer Lecture Notes in Artificial Intelligence.
12. R. Küsters and A. Borgida. What's in an attribute? Consequences for the least common subsumer. *Journal of Artificial Intelligence Research*, 14: 167–203, 2001.
13. R. Küsters and R. Molitor. Computing least common subsumers in $\mathcal{ALEN}$. In *Proc. of IJCAI'01*. Morgan Kaufmann, 2001. To appear.
14. R. Küsters and R. Molitor. Computing most specific concepts in description logics with existential restrictions. Technical Report LTCS-00-05. See http://www-lti.informatik.rwth-aachen.de/Forschung/Reports.html.
15. B. Nebel. Reasoning and Revision in Hybrid Representation Systems. LNAI 422, Springer-Verlag, 1990.
16. A. Schaerf. On the complexity of the instance checking problem in concept languages with existential quantification. *Journal of Intelligent Information Systems*, 2:265-278, 1993.
17. L. von Wedel and W. Marquardt. ROME: A repository to support the integration of models over the lifecycle of model-based engineering processes. In *Proc. of ESCAPE-10*, 2000.

Bayesian Learning and Evolutionary Parameter Optimization

Thomas Ragg

Institut für Logik, Komplexität und Deduktionssysteme,
Universität Karlsruhe, D-76131 Karlsruhe,
ragg@ira.uka.de http://i11www.ira.uka.de

Abstract. In this paper I want to argue that the combination of evolutionary algorithms and neural networks can be fruitful in several ways. When estimating a functional relationship on the basis of empirical data we face three basic problems. Firstly, we have to deal with noisy and finite-sized data sets which is usually done be regularization techniques, for example Bayesian learning. Secondly, for many applications we need to encode the problem by features and have to decide which and how many of them to use. Bearing in mind the empty space phenomenon, it is often an advantage to select few features and estimate a non-linear function in a low-dimensional space. Thirdly, if we have trained several networks, we are left with the problem of model selection. These problems can be tackled by integrating several stochastic methods into an evolutionary search algorithm. The search can be designed such that it explores the parameter space to find regions corresponding to networks with a high posterior probability of being a model for the process, that generated the data. The benefits of the approach are demonstrated on a regression and a classification problem.

1 Learning Based on Empirical Data

If one wants to learn a functional relationship from empirical data, then the goal of network training is to recognize a structure in the data or an underlying process and to generalize this knowledge to former unknown data points. Bishop characterizes the goal of network training as *building a statistical model of the process which generates the data* [Bishop, 1995, Ripley., 1996, Hertz *et al.*, 1991]. If we approximate the data-generating process the error over all former unseen patterns will be minimal.

1.1 Regularization

A sound discussion on a theoretic level of the generalization problem can be found in [Vapnik, 1982, Vapnik, 1995]. Vapnik states that the problem of density estimation based on empirical data is ill posed, i.e., small changes in the learning situation can result in a totally different model; for example little distortions in the target data. The theory of regularization shows that instead of minimizing

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 48–62, 2001.

the difference between the target data and the output of the model, a regularized error function

$$E = E_D + \lambda E_R \tag{1}$$

should be minimized where E_D is the error on the data and λ is a weighting factor. E_R is an additional term that measures the complexity of the model; for example the often used weight decay regularizer in neural network training [Krogh & Hertz, 1992, Bishop, 1995]. Thus, regularization is not an optional possibility, but a fundamental technique [Ramsay & Silverman, 1997]. One crucial problem is to determine the weighting factor λ. In case of neural networks, its optimal value depends on the size of the network, i.e., the number of weights, the weight initialization, as well as the patterns used for training, and the noise in the data. Often this value is determined by cross-validation which is clearly suboptimal for the reasons just given. Furthermore, an additional data set is necessary to determine this parameter experimentally in a series of training runs; i.e., additional computational effort has to be investigated to fit the parameter λ, while on the other hand less data points are left for the actual training process.

Adjusting this value properly has been solved for neural networks by using a Bayesian learning algorithm. It was introduced by MacKay [MacKay, 1992a] and provides an elegant theory to prevent neural networks from overfitting by determining λ during the training process without the necessity of additional validation data. Still different models will be found depending on the initialization and the network topology (Fig. 2a), which can then be compared on basis of their posterior probability. A short review of Bayesian learning for neural networks as used in this work is given in section 2 (see [MacKay, 1992a, Bishop, 1995] or [Gutjahr, 1999, Ragg, 2000] for a detailed discussion on Bayesian learning).

1.2 Curse of Dimensionality

In practical applications only a limited amount of data is available for determining the parameters of a model. The dimensionality of the input vector must be in a sensible relation to the number of data points. By adding more features the information content of the input about the target increases, but at the same time the number of data points per dimension decreases in order to determine the parameters. That means that the class of functions in which the solution is searched increases greatly with every additional component. This problem is called the *curse of dimensionality* [White, 1989, Bishop, 1995] or the *empty space phenomenon* [Scott & Thompson, 1983, Silverman, 1986]. A consequence is that renunciation of supplementary information leads to a more precise approximation of the underlying process in a low-dimensional input space. Table 1 relates the dimensionality of the input vector to the number of patterns that is necessary to approximate a simple non-linear function [Silverman, 1986]. Note that for most of the known benchmark problems in machine learning databases, these values are not reached.

A further aspect which is to be considered is the relation between the number of input features used and the size of the regularization term. Adding features results in a higher value of the regularization term, since the network structure

Table 1. Number of patterns that are necessary to estimate a multivariate normal distribution in the origin with a mean square error below 0.1

dimensions	number of patterns	dimensions	number of patterns
1	4	6	2.790
2	19	7	10.700
3	67	8	43.700
4	223	9	187.000
5	768	10	842.000

grows. That is, if the additional features contribute only little information, they nevertheless lead to a stronger punishment of the complexity. In practical experiment one observes that learning in high dimensional input spaces often leads to almost linear solutions (cf. Fig.4, see also [Ragg, 2000]).

1.3 Model Selection

It is common practice to train several networks and then select the best one according to the performance on an independent validation set. This procedure has the disadvantages as already noted above in the context of determining the weighting parameter λ.

Bayesian learning for neural networks does not only allow to regularize the model automatically but also provides a quality measure, the so called model evidence. On this basis it is possible to select the model with the highest evidence from a set of trained networks. This procedure is described below in greater detail.

Forming a committee of networks is another approach to overcome these drawbacks and to avoid favouring one model while discarding all others. Several methods of forming committees were suggested in recent years. An overview is given in the book *Combining Artificial Neural Nets* [Sharkey, 1999].

In this paper we focus on the optimization of a single model and using the model evidence as selection criteria. A committee approach that combines proper regularization with Bayesian learning and a evolutionary search for (stochastically) independent networks is described in [Ragg, 2000]. Note that independence of committee members is crucial to further improvement of the generalization performance.

2 Neural Networks and Bayesian Learning

The goal of network training is to estimate a functional relationship based on the given data. To minimizes the expected risk, i.e., the generalization error, the data-generating process should be approximated. This process is usually described in terms of the joint probability $p(\mathbf{x}, t) = p(t|\mathbf{x})p(\mathbf{x})$ in input-target space. To make a prediction for former, unseen values of $\mathbf{x}$ the conditional probability $p(t|\mathbf{x})$ is of interest. The neural network is modelling this density.

Different assumptions about this density lead to different error functions. For example, assuming that the target data is generated from a function with additive Gaussian noise, leads to the well known mean square error. In the following I will focus on regression problems using the mean square error function. The theory can although be applied to classification problems as well. For an excellent overview, see [Bishop, 1995].

The application of Bayesian learning to neural networks was introduced by MacKay as a statistical approach to avoid overfitting, [MacKay, 1992a, Bishop, 1995]. In contrast to the sampling theory, where the free parameters of probability must be attached by frequencies of samples drawn from a distribution, the Bayesian statistics uses a priori probabilities to describe degrees of beliefs in parameters. The main idea is to combine the sample information and the prior belief to form the posterior probability of the parameters.

Given the training data D we want to find the 'best' model $M(\Theta)$ with parameter vector Θ. This is expressed in Bayes' theorem as follows

$$p(\Theta|D) = \frac{p(D|\Theta)\ p(\Theta)}{p(D)} \tag{2}$$

The best model maximizes the posterior probability, i.e., we want to find Θ^* such that

$$p(\Theta^*|D) \geq p(\Theta|D) \ \ \forall\, \Theta.$$

In case of neural networks the parameter vector might consist of the weight vector $\mathbf{w}$, the weighting factor λ as well as the topology of the network. In general, we cannot determine all these parameters at once. For that, the Bayesian framework proceeds in a hierarchical fashion. While keeping all other parameters fixed we search on the first level for an optimal weight vector, on the second level for optimal weighting coefficients and on the third level for the appropriate network topology. Comparing network topology is often done manually by the developer. In this paper we suggest to intertwine the Bayesian learning approach with an evolutionary search procedure to find an optimal topology and select input features in an efficient way. In the following the three levels are described in more detail.

2.1 The Likelihood

If a training set $D = \{(x_1, t_1), \ldots, (x_N, t_N)\}$ of N input-target pairs is given and if we assume that the target data is generated from a function with additive Gaussian noise, the probability of observing t_m is given by

$$p(t_m|x_m, \mathbf{w}, \beta) = \frac{\beta}{2\pi} \exp(-\frac{\beta}{2}\{y(x_m, \mathbf{w}) - t_m\}^2), \tag{3}$$

where $y(x_m, \mathbf{w})$ denotes the output of the neural network with weight vector $\mathbf{w}$ and β controls the noise in the data, i.e., $1/\beta$ is just the variance of the noise. Provided the data points are drawn independently, the probability of the data, called the likelihood is

$$p(D|\mathbf{w},\beta) = \prod_{m=1}^{N} p(t_m|x_m,\mathbf{w},\beta)$$
$$= \frac{1}{Z_D(\beta)} \exp\left(-\frac{\beta}{2}\sum_{m=1}^{N}\{y(x_m,\mathbf{w}) - t_m\}^2\right) \quad (4)$$

where $Z_D(\beta) = (\frac{2\pi}{\beta})^{N/2}$ is the normalizing constant. It follows that the sum-of-squares error of the data $E_D(\mathbf{w}) = \frac{1}{2}\sum\{y(x_m,\mathbf{w}) - t_m\}^2$ just expresses the likelihood function.

2.2 The Prior Probability

In the simplest case of Bayesian learning the prior probability distribution of the weights is assumed to be Gaussian with zero mean. This restricts the complexity of the neural network by searching for small weight values. By denoting $\frac{1}{\alpha}$ as the variance of the Gaussian the prior is of the form

$$p(\mathbf{w}|\alpha) = \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W)$$
$$= \frac{1}{Z_W(\alpha)} \exp(-\frac{\alpha}{2}\sum_{j=1}^{W} w_j^2) \quad (5)$$

where $Z_W(\alpha) = (\frac{2\pi}{\alpha})^{W/2}$ is the normalization factor.

2.3 The Posterior Distribution of the Weights

Due to Bayes' rule (2) we get from (4) and (5) the posterior weight distribution

$$p(\mathbf{w}|D,\alpha,\beta) = \frac{1}{Z_S(\alpha,\beta)} \exp\{-\beta E_D - \alpha E_W\} \quad (6)$$

Instead of maximizing the posterior it is equivalent but simpler to minimize the exponent. This is exactly learning in the sense of equation (1) with $\lambda = \frac{\alpha}{\beta}$. Usually fast learning algorithms like Scaled Conjugate Gradient [Møller, 1993] or Rprop [Riedmiller, 1994] are used to find a local minimum of the error function.

2.4 The Evidence for α and β

On the first level of the Bayesian framework a method was developed to optimize the weights where the so-called hyperparameters α and β were assumed to be known. Due to the Bayesian approach we are able to determine the hyperparameters automatically during the training process. This can be done by maximizing the probability of α and β given the data.

$$p(\alpha,\beta|D) = \frac{p(D|\alpha,\beta)p(\alpha,\beta)}{p(D)} \quad (7)$$

The prior for α and β are usually assumed to be constant in order to give equal importance to all possible values [Bishop, 1995]. Such priors are called to be

non-informative. The big disadvantage of uniform priors is that they are not invariant under reparametrization. A widely used method to determine non-informative priors that are invariant under reparametrization is presented by Jeffreys, [Jeffreys, 1961]. Gutjahr calculated Jeffreys prior for the hyperparameters α and β to $p(\alpha) = \frac{1}{\alpha}$ resp. $p(\beta) = \frac{1}{\beta}$ (cf. [Gutjahr, 1998], and [Gutjahr, 1999] for an extensive discussion).

In order to optimize the hyperparameters during the training process, we have to maximize the probability of α and β given the data according to (7). Note that it is assumed, that the weight vector is optimized, i.e., the network was trained to a local minimum of the error function. This is denoted in the following with E_W^{MP}, E_D^{MP}, etc.[1] With Jeffreys priors for α and β we get

$$\begin{aligned}\log p(\alpha, \beta|D) &\propto \log p(D|\alpha, \beta) + \log p(\alpha, \beta) \\ &= -\alpha E_W^{MP} - \beta E_D^{MP} - \frac{1}{2}\log\det(\mathbf{A}) \\ &\quad + \frac{W}{2}\log\alpha + \frac{N}{2}\log\beta - \frac{N}{2}\log(2\pi) \\ &\quad - \log\alpha - \log\beta \end{aligned} \tag{8}$$

where $\mathbf{A}$ is the Hessian of the error function and we assumed the hyperparameters to be independent. The formula is basically the same as in [Bishop, 1995] except for the last two summands (cf. [Bishop, 1995] or [Gutjahr, 1999, Ragg, 2000] for details of the computation). We calculate the partial derivatives of (8) with respect to α and β and determine the optimal values of the hyperparameters by setting the results to zero. This provides the following update rules for α and β

$$\alpha^{new} = \frac{1}{2E_W^{MP}} \sum_{i=1}^{W} \frac{\lambda_i}{\lambda_i + \alpha} - \frac{1}{E_W^{MP}} \tag{9}$$

and

$$\beta^{new} = \frac{1}{2E_D^{MP}} (N - \sum_{i=1}^{W} \frac{\lambda_i}{\lambda_i + \alpha}) - \frac{1}{E_D^{MP}} \tag{10}$$

In comparison to the formulas given in [Bishop, 1995] we see that these update rules produce smaller values for the hyperparameters. This does not automatically mean that λ will also get smaller, because the latter depends on the relative values of α and β.

In a practical implementation we have to find optimal values for α and β as well as for $\mathbf{w}$ at the same time. This is solved by a iterative algorithm that periodically re-estimates α and β according to equation (9) and equation (10), after a minimum of the current error function was reached. Note that when re-estimating the hyperparamters the error function changes. Figure 1 illustrates the iterative algorithm, which has similarities with the basic principle of the EM-algorithm.

[1] MP means most probable

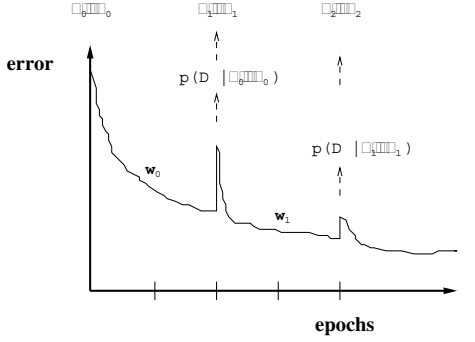


Fig. 1. The figure shows the error curve during training and the effects of the update of the hyperparameters. Initially the are initialized with α_0 and β_0, than the weight vector is optimized by gradient descent. If a minimum of the current error function is reached the evidence $p(D|\alpha_0, \beta_0)$ is computed and the update is performed. Note that it takes at least two updates until we have with $p(D|\alpha_1, \beta_1)$ a value for the evidence that is based on hyperparameters that were computed by the Bayesian approach.

A great advantage of the Bayesian approach is that a large number of regularization coefficients can be used since they are determined automatically. Weights can be grouped together where each group gets an own hyperparameter α_k. That is, the error function is changed to

$$E = \beta \cdot \frac{1}{2} \sum_{n=1}^{N} (y(x_n; \mathbf{w}) - t_n)^2 + \sum_{k=1}^{G} \left(\alpha_k \cdot \frac{1}{2} \sum_{w \in \mathcal{W}_k} w^2 \right) \tag{11}$$

The extension of the Bayesian framework to several weight groups is straightforward [MacKay, 1992b, Thodberg, 1993, Nautze & Gutjahr, 1997]. It is common to put all the weights between two layers in one group [Bishop, 1995, Gutjahr, 1999]. In case of small data sets or in case of networks with few hidden units it is sometimes of advantage to use only one weight group as in the standard weight decay approach, since only one hyperparameter controlling a distribution has to be estimated on the basis of few weights.

2.5 The Model Evidence

On the third level we can compare different models, for example, networks with different topologies. Using Bayes' rule we can write the posterior probability of a model $\mathcal{H}$ as

$$P(\mathcal{H}|D) = \frac{p(D|\mathcal{H}) \cdot P(\mathcal{H})}{p(D)}$$

If we assign the same prior to every model, than it is sufficient to evaluate the quantity $p(D|\mathcal{H})$, which is called the evidence for $\mathcal{H}$ [MacKay, 1992a, Bishop, 1995]. Marginalizing over α and β provides

$$p(D|\mathcal{H}) = \int \int p(D|\alpha, \beta, \mathcal{H}) p(\alpha, \beta|\mathcal{H}) \, d\alpha \, d\beta$$

The first factor on the right hand side is just the evidence for α and β from the previous level. Integration over α and β is usually done by approximating their posterior distributions on the second level by Gaussians. See [Bishop, 1995] and especially [Gutjahr, 1999] for details. The logarithm of $p(D|\mathcal{H})$ is then given as

$$\begin{aligned}\ln p(D|\mathcal{H}) = &-\beta_{opt}E_D - \alpha_{opt}E_W - \frac{1}{2}\ln\,\det(\mathbf{A}) \\ &- \frac{N}{2}\ln(2\pi) + \frac{N}{2}\ln(\beta_{opt}) + \frac{W}{2}\ln(\alpha_{opt}) \\ &+ 2\ln\sqrt{2\pi} + \frac{1}{2}\ln\left(\frac{2}{\gamma}\right) + \frac{1}{2}\ln\left(\frac{2}{N-\gamma}\right). \end{aligned} \tag{12}$$

Note that the evidence depends on the determinant of the Hessian,$\det(\mathbf{A})$; i.e., the product of its eigenvalues. This makes the result more sensitive to little deviations caused by approximations. Nonetheless, the evidence is usually negatively correlated with the generalization error [MacKay, 1992b, Thodberg, 1993, Ragg & Gutjahr, 1998] giving us a hint which networks to prefer.

It is possible to exploit this correlation even further by intertwining a search procedure with the iterative Bayesian learning algorithm. By using a population of networks every iteration step can be considered as a generation of an evolutionary algorithm. Instead of re-training all networks, the ones with higher evidence are allowed to provide more offsprings on expense of the others. That means in figure 1 that after two re-estimation steps the values of the evidence for the different networks are compared and the ones with lower values are discarded. In this way the training process is restricted to favourable areas of the search space, such that the resulting networks have higher evidence compared to the standard training procedure [Ragg, 2000]. In the following I will describe the basic evolutionary algorithm and its operators.

2.6 Evolutionary Search

The concept of solving optimization problems by means of evolutionary strategies was developed in the 1960s by Rechenberg and Schwefel [Rechenberg, 1994, Schwefel, 1995], and was since then used for a variety of problems including the optimization of neural networks [Alander, 1996]. The approach pursued at our institute was to use evolution for topology optimization and combine it with learning by gradient descent [Braun & Ragg, 1996, Braun, 1997]. Recently, we showed that the evolutionary search for sensible topologies and the iterative Bayesian learning framework can be intertwined by using the model evidence, i.e., its posterior probability, as fitness value [Ragg, 2000]. Figures 3 and 4 show the dependency of the evidence from the number of hidden units and the number of input features. An evolutionary algorithm will tend to find local maxima in the shown fitness landscapes.

Using an evolutionary algorithm as framework for searching an optimal solution with respect to the fitness function as described below has several advantages: As every heuristic search algorithm it has to handle the dilemma between exploitation and exploration. This is done by parallelizing the search using a

population of networks and stochastic selection of the parents each generation. This explorative search is biased towards exploitation by preferring fitter parents. Furthermore, the algorithm is scalable with regard to computing time and performance, i.e., a larger population allows for a more explorative search.

Incremental elimination of input units does in general not lead to an optimal subset of features [Fukunaga, 1990, Bishop, 1995]. Adding input units during the optimization process which maximize the information content can overcome these local minima with little computational effort [Ragg & Gutjahr, 1997]. The underlying framework may be summarized as follows:

Pre-evolution: Initially, a population of $\mu \geq 1$ networks is created by copying a maximal topology that is chosen to restrict the search space. By randomly deleting units from the networks and initializing the weights according to the assumptions of the Bayesian framework (cf. section 2.2), we start the search from different points. These networks are trained by Bayesian learning gradient descent as described in section 2. Each parent is assigned its model evidence as fitness value.

Selection: Offsprings are generated after every second re-estimation of the hyperparameters. Each generation $\lambda \geq \mu$ offsprings are generated by first sorting the parents into a list based on their fitness, and then selecting λ individuals with replacement according to a selection function of type x^y. The parameter y is a prefer-factor. If $y \approx 1$, then all elements have a similar probability to be selected. If y gets larger, the probability to select elements from the beginning of the list grows accordingly. This mechanism allows us to select fitter networks with higher probability. The offspring are identical copies of the parents including the values for the hyperparameters (Lamarckism). Note that we use a so called (μ, λ)-strategy [Schwefel, 1995]. This is sensible because the search is intertwined with the Bayesian framework. An elitist strategy, i.e., a $(\mu + \lambda)$-strategy, would not serve our purpose since parent networks are not retrained. They would be at a disadvantage against offsprings because they run through fewer re-estimation cycles.

Mutation: To exploit the area around a parent model noise is added to the weights and hyperparameters of the offsprings. The random values are drawn from a normal distribution with zero mean and variance 10^{-3}) and scaled by the value of the component, i.e., $w_i \rightarrow w_i + w_i * rand()$. This is useful, because several approximations are made in the training process.

To change the network structure in a sensible way, we need to have a criterion to sort the units by saliency and delete low-saliency units. On the other hand, in case of addition of units, we should increase the network size only by important items. Since the degree of freedom is controlled by the weight-decay regularizer, simple and fast computable heuristics are used to optimize the number of hidden units. Weights are not removed to avoid small weight groups, which would interfere with the estimation of the hyperparameters in the Bayesian framework. Since a regularizer is used, it is not necessary to use advanced (computational

intensive) methods that use the Hessian since they do not perform better than simple pruning techniques in this case [Ragg *et al.*, 1997]. Thus, hidden units are sorted with respect to the size of the sum of their weights. Adding hidden units is done randomly. The most computational effort is used to optimize the input structure The input units are sorted with respect to the information content $I(X;Y)$ (mutual information) of the resulting input vector about the target: If

$$X = (X_1, \cdots, X_{k-1}, X_{k+1}, \cdots, X_{i-1}, X_i, X_{i+1}, \cdots, X_d)$$

is the actual input vector of our neural network than we define

$$X_{(i)} := (X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_d)$$

and

$$X^{(k)} := (X_1, \ldots, X_{k-1}, X_k, X_{k+1}, \ldots, X_d).$$

This means that $X_{(i)}$ removes the i-th component of X whereas $X^{(k)}$ adds the k-th component to X. Every time an input component is to be deleted from resp. added to our actual input vector X we calculate $I(X_{(i)};Y)$ as well as $I(X^{(k)};Y)$ where i, k take all possible numbers. See [Ragg & Gutjahr, 1997, Ragg, 2000] for details of the computation. A nice introduction into information theory can be found in [Cover & Thomas, 1991], An overview on probability density estimation procedures is given by [Silverman, 1986]. They are used to estimate the mutual information between the input vector and the target. Based on this calculation, we sort the resulting subsets of input variables by the following criteria:

Remove: $X_i \prec X_j :\Longleftrightarrow I(X_{(i)};Y) > I(X_{(j)};Y)$

Insert: $X^k \prec X^l :\Longleftrightarrow I(X^{(k)};Y) > I(X^{(l)};Y)$

For each selected parent one input unit and/or one hidden unit is mutated with a certain probability. The candidates for input unit mutation are chosen from the sorted list according to a selection function (as above) where y takes a large value, e.g., $y \geq 2$, i.e, the first 25% of the list are selected at least as often as the last 75%.

Optimization and Evaluation: The resulting offspring networks are then trained as described in section 2. The hyperparameters are re-estimated two times during the optimization phase. The reason for that is, that the Bayesian framework allows us to compute the evidence of the network after the error was minimized. The evidence refers to the hyperparameters which were used during this cycle. That means, it is not until the second iteration that we can be sure that the computation of the evidence is based on a set of hyperparameters that were estimated in an optimal way (cf. fig.1). At last the model evidence is assigned to the network as fitness value.

Survival: The loop is repeated until a user-defined stop criterion is fulfilled, e.g., a fixed number of generations, or until the population collapsed into a single search point. The algorithm just presented favours in every generation networks with higher evidence. That is, the search is with each generation less and less explorative and will finally find a (local) optimum of the evidence in parameter space.

3 Experimental Results

Two problems are used here to evaluate the approach - a noisy regression problem and a classification task from the UCI repository. Results on more problems are described in [Ragg, 2000].

Noisy sine regression: The data for this benchmark was generated by adding Gaussian noise to a sine function (Figure 2a). 40 data points were used for training the model while 100 additional data points (from the underlying process) were used to compute the test error. Figure 2b shows several models for this data set, two of them trained with Bayesian learning, which show sensible yet still different approximations of the underlying process. These dependency of the results on the initialization and the network topology is visualized in figure 3. Several thousand networks were trained to establish this relationship. We clearly observe that there is an optimal network topology with about 15 to 20 hidden units. The correlation between the evidence and the test error was $\rho = -0.35$. An evolutionary optimization using the evidence as fitness value will now tend to find a maximum of the model evidence in parameter space which can clearly be seen in figure 3b. The average error of all trained networks is 0.04 ± 0.08 which reduces to 0.014 ± 0.012 if we train 100 networks with randomly chosen topology and select the model with highest evidence. Evolutionary optimization further reduces the test error to 0.011 ± 0.003. All improvements are significant according to the t-test (threshold $t_{0.95;10} = 1.82$); see [Büning & Trenkler, 1994] for statistical test procedures.

Thyroid classification: In the problem described above the input consisted of only one neuron, i.e. feature selection was not necessary. The thyroid classification problem is form the UCI repository and contains 21 input features. About 7200 patterns are available, from which we used only 600 for training the

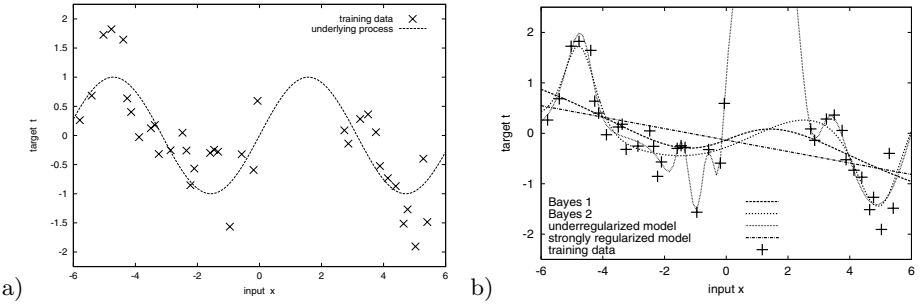


Fig. 2. The figure shows the data for the benchmark regression problem. a) The underlying process is just the sine function $sin(x)$. The target data was generated by adding Gaussian noise with variance 0.4 and zero mean. Note that there is an interval, where no data is available. b) Output functions of 4 models are plotted: An overregularized network, an underregularized network and two proper regularized networks trained with Bayesian learning.

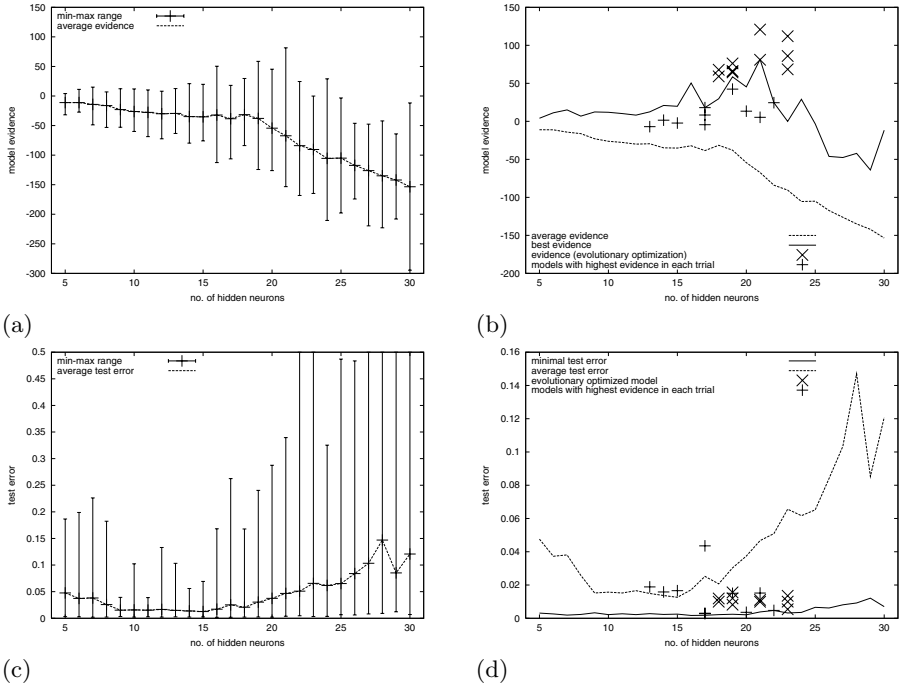


Fig. 3. The figure shows the dependency of the evidence from the number of hidden units and the number of input features, which was determined by the training of several thousands of networks. a) Average, minimal and maximal evidence. The networks with the highest evidence have between 16 and 24 neurons in the hidden layer. b) Range of average and maximal evidence. The crosses (+) correspond to the networks with the highest evidence (of ten trials of training 50 networks). The other symbols (×) mark the results for the evolutionary optimization. They are all concentrated in the area of the maximum. c) Average, minimal and maximal test error. d) Range of average and minimal test error. Symbols used as in (b)

networks and the other 6600 solely to estimate the generalization performance. The task is to classify a malfunction of the thyroid (under, over or normal). According to table 1 we should not exceed an input dimensionality of 5 or 6 features, if we want to learn a non-linear relationship. Figure 4a shows clearly a region with maximal evidence (left bottom corner) which is highly correlated with the generalization error ($\rho = -0.90$). The key problem here is to find an optimal subset of input features. Note that this is a combinatorical problem (different from optimizing the number of hidden features). For the given fitness landscapes the proposed evolutionary algorithm performs a gradient ascent to the region with maximal evidence. This is possible since Bayesian learning regularizes the networks properly and the mutation of input units based on mutual information leads so sensible subsets of features with a high information content. The error rate of all trained networks is $2.8 \pm 0.01\%$ which reduces to $2.6 \pm 0.1\%$ if we train 100 networks with randomly chosen topology and select the model

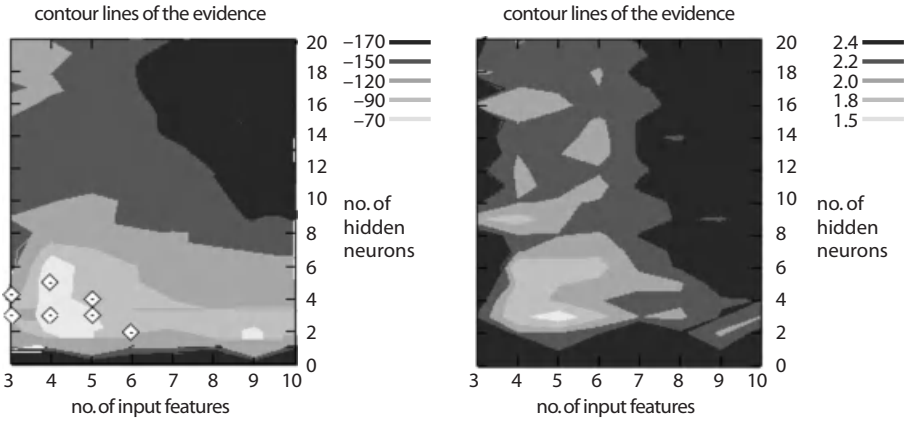


Fig. 4. The figure shows the dependency of the evidence resp. the generalization error from the number of hidden units and the input features. About 10000 networks (with random topology) were trained to establish this dependency. Only 10 input features are shown for a better visualization. a) Model evidence over parameter space. A clear maximum is observable for 4 to 6 input features and 2 to 6 hidden neurons, which can easily be found by a evolutionary search procedure if we start with random search points in this parameter space. The rhombuses mark the results of 10 evolutionary optimization runs, from which some lead to identical network structure. All models group in the region of high evidence. b) Generalization error over parameter space. The evidence is highly correlated with the generalization error ($\rho = -0.90$).

with highest evidence. Evolutionary optimization further reduces the test error to $1.8 \pm 0.3\%$. All improvements are significant according to the t-test.

4 Conclusions

In this paper an approach was presented that integrates several important steps of neural network design into an evolutionary optimization procedure. This method was primarily aimed at developing models for tasks where the data is noisy and limited. Bearing in mind the curse of dimensionality we should select few features with a high information content, thus approximating the function being searched in a low-dimensional space. Furthermore, it is difficult to decide which part of the data should be used for training. This problem can be tackled by integrating over several samples of the data set. Moreover, in a practical application we also face the problem of model selection. That is, having trained several networks, we would like to select one that performs significantly better than average.

We started out be reviewing the some problems that arise when trying to estimate a functional relationship with neural networks. Learning from empirical data is an ill posed problem making it necessary to use a proper regularization method, e.g., weight decay. A crucial problem is to determine the weighting factor of the regularizer. This can best be done by applying a Bayesian learning

algorithm, because it adapts this parameter automatically without the necessity of additional validation data. Furthermore, it provides a quality measure, i.e., the model evidence, which can be used for model comparison. Furthermore, the evidence can serve as the optimization criterion of an evolutionary algorithm (cf. [Ragg & Gutjahr, 1998]). By this, the trial and error search for models with high posterior probability can be replaced by a more efficient search procedure. Moreover, feature selection and topology optimization can be intertwined with the network training such that a local maximum of the fitness function over parameter space can be found. For two problems it was shown that the optimization procedures finds regions with higher model evidence thus lowering the risk of choosing a wrong model and improving significantly the generalization performance of the models.

Acknowledgments. This work was supported by the Deutsche Forschungsgemeinschaft (DFG), ME 672/10-1, 'Integrierte Entwicklung von Komitees neuronaler Netze'.

References

[Alander, 1996] Alander, J. T. An Indexed Bibliography of Genetic Algorithms and Neural Networks. Technical Report 94-1-NN, Department of Information Technology and Production Economics, University of Vaasa, 1996.

[Bishop, 1995] Bishop, C. M. *Neural Networks for Pattern Recognition.* Oxford Press, 1995.

[Braun & Ragg, 1996] Braun, H. and Ragg, T. ENZO – Evolution of Neural Networks, User Manual and Implementation Guide, http://i11www.ira.uka.de. Technical Report 21/96, Universität Karlsruhe, 1996.

[Braun, 1997] Braun, H. *Neuronale Netze: Optimierung durch Lernen und Evolution.* Springer, Heidelberg, 1997.

[Büning & Trenkler, 1994] Büning, H. and Trenkler, G. *Nichtparametrische statistische Methoden.* de Gruyter, 1994.

[Cover & Thomas, 1991] Cover, T. and Thomas, J. *Elements of Information Theory.* Wiley Series in Telecommunications. John Wiley & Sons, 1991.

[Fukunaga, 1990] Fukunaga, K. *Introduction to Statistical Pattern Recognition.* Academic Press, 1990.

[Gutjahr, 1998] Gutjahr, S. Improving the determination of the hyperparameters in bayesian learning. In Downs, T., Frean, M., and Gallagher, M., editors, *Proceedings of the Ninth Australian Conference on Neural Networks (ACNN 98)*, pages 114–118, Brisbane, Australien, 1998.

[Gutjahr, 1999] Gutjahr, S. *Optimierung Neuronaler Netze mit der Bayes'schen Methode.* Dissertation, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 1999.

[Hertz *et al.*, 1991] Hertz, J., Krough, A., and Palmer, R. G. *Introduction to the theory of neural computation*, volume 1 of *Santa Fe Institute, Studies in the sciences of complexity, lecture notes.* Addison-Wesley, 1991.

[Jeffreys, 1961] Jeffreys, H. *Theory of Probability.* Oxford University Press, 1961.

[Krogh & Hertz, 1992] Krogh, A. and Hertz, J. A Simple Weight Decay Can Improve Generalisation. In *Advances in Neural Information Processing 4*, pages 950–958, 1992.

[MacKay, 1992a] MacKay, D. J. C. Bayesian interpolation. *Neural Computation*, 4(3):415–447, 1992.

[MacKay, 1992b] MacKay, D. J. C. A practical bayesian framework for backpropagation networks. *Neural Computation*, 4(3):448–472, 1992.

[Møller, 1993] Møller, M. A Scaled Conjugate Gradient Algorithm for fast Supervised Learning. *Neural Networks*, 6:525–533, 1993.

[Nautze & Gutjahr, 1997] Nautze, C. and Gutjahr, S. Extended bayesian learning. In *Proceedings of ESANN 97, European Symposium on Artificial neural networks, Bruges*, pages 321–326, 1997.

[Ragg & Gutjahr, 1997] Ragg, T. and Gutjahr, S. Automatic Determination of Optimal Network Topologies based on Information Theory and Evolution. In *IEEE, Proceedings of the 23rd EUROMICRO Conference 1997*, pages 549–555, 1997.

[Ragg & Gutjahr, 1998] Ragg, T. and Gutjahr, S. Optimizing the Evidence – with an application to Time Series Prediction. In *Proceedings of the ICANN 1998, Sweden*, Perspectives in Neural Computing, pages 275–280. Springer, 1998.

[Ragg *et al.*, 1997] Ragg, T., Braun, H., and Landsberg, H. A Comparative Study of Neural Network Optimization Techniques. In *Lecture Notes in Computer Science, Proceedings of the ICANNGA 1997, Norwich, UK*, pages 343–347. Springer, 1997.

[Ragg, 2000] Ragg, T. *Problemlösung durch Komitees neuronaler Netze*. Dissertation, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 2000.

[Ramsay & Silverman, 1997] Ramsay, J. O. and Silverman, B. *Functional data analysis*. Springer, 1997.

[Rechenberg, 1994] Rechenberg, I. *Evolutionsstrategie '94*. Frommann-Holzboog Verlag, Stuttgart, 1994.

[Riedmiller, 1994] Riedmiller, M. Advanced supervised learning in multi-layer perceptrons - from backpropagation to adaptive learning algorithms. *Int. Journal of Computer Standards and Interfaces*, 16:265–278, 1994. Special Issue on Neural Networks.

[Ripley., 1996] Ripley., B. D. *Pattern recognition and neural networks*. Cambridge University Press, 1996.

[Schwefel, 1995] Schwefel, H.-P. *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology Series. John Wiley & Sons, 1995.

[Scott & Thompson, 1983] Scott, D. and Thompson, J. Probability density estimation in higher dimensions. In Gentle, J., editor, *Computer Science and Statistics: Proceedings of the Fifteenth Symposium on the Interface*, pages 173–179. 1983.

[Sharkey, 1999] Sharkey, A. J. Multi-Net Systems. In Sharkey, A. J., editor, *Combining Artificial Neural Nets*, pages 1–30. Springer, 1999.

[Silverman, 1986] Silverman, B. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1986.

[Thodberg, 1993] Thodberg, H. H. Ace of bayes: Applications of neural networks with pruning. Technical Report 1132E, Danish Meat Research Institute, 1993.

[Vapnik, 1982] Vapnik, V. *Estimation of Dependences Based on Empirical Data*. Springer, 1982.

[Vapnik, 1995] Vapnik, V. *The Nature of Statistical Learning Theory*. Springer, 1995.

[White, 1989] White, H. Learning in artificial neural networks: a statistical perspective. *Neural Computation*, 1:425–464, 1989.

Abductive Partial Order Planning with Dependent Fluents

Liviu Badea and Doina Tilivea

AI Lab, National Institute for Research and Development in Informatics
8-10 Averescu Blvd., Bucharest, Romania
badea@ici.ro

Abstract. Our query planning application for system integration requires a backward partial-order planner able to deal with non-ground plans in the presence of state constraints. While many partial-order planners exist for the case of independent fluents, partial-order planning with *dependent fluents* is a significantly more complex problem, which we tackle in an abductive event calculus framework. We show that existing abductive procedures have *non-minimality* problems that are significant especially in our planning domain and propose an improved abductive procedure to alleviate these problems. We also describe a general transformation from an abductive framework to *Constraint Handling Rules (CHRs)*, which can be used to obtain an efficient implementation.

1 Introduction and Motivation

The integration of hybrid modules, components and software systems, possibly developed by different software providers, is a notoriously difficult task, involving various extremely complex technical issues (such as distribution, different programming languages, environments and even operating systems) as well as conceptual problems (such as different data models, semantic mismatches, etc.).

Solving the conceptual problems requires the development of a common, explicit, declarative knowledge model of the systems to be integrated. Such a model should be used (by a so-called *mediator*) not only during the development of the integrated system, but also during runtime, when it can be manipulated by an intelligent *query planning* agent to solve problems that could not have been solved by any of the specific information sources alone and might even not have been foreseeable by the system integrator. Since in most realistic applications, the state of the databases or of the procedural components changes as a problem is being solved, we shall describe the services offered by such procedural applications and the database updates as *actions*.

The *query planner* of the mediator transforms a user query into a partially ordered sequence of information-source specific queries, updates and calls to application interfaces that solve the query. The main requirements which our system integration application imposes on the query planner are the following:

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 63–77, 2001.

- It should be a *partial-order planner* in order to take advantage of the intrinsic distributed nature of the integrated system.
- As the information content of the sources (for example databases) can be quite large, forward propagation of the initial state of the sources is impossible in practice. We therefore need to develop a *backward* planner, which would ensure minimal source accesses with maximally specific queries.
- The planner should reason with *arbitrary logical constraints* between state predicates (*dependent* fluents).
- It should also be able to manipulate *plans with variables*. UCPOP-like planners can do this, but UCPOP only deals with *independent* fluents.

Also, although very fast general purpose planners like *Graphplan* and *SATPLAN* are currently available, these are not usable in our system integration application, where generating a grounding of the planning problem is inconceivable (a database may contain an enormous number of different constants), while both *SATPLAN* and *Graphplan* generate a grounding of the planning problem. The same holds for recent planners based on Answer Set Programming developed in the Logic Programming community, which use efficient *propositional* answer set generators, like *dlv* or *smodels*.

Unfortunately, there is no implemented planner available with the characteristics required by our system integration application. In this paper we describe the construction of such a planner.

The following simple example illustrates the type of problems we are dealing with. Assume that the dean of a university plans to assign a high responsibility course (this course being assignable only to faculty members). This can be done by applying action assign_course having effect course_assigned and no explicit preconditions. Assume there exist constraints, such as that this course cannot be assigned to a person who is not a professor

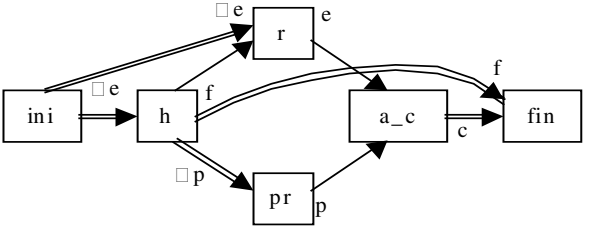
$$\leftarrow \text{holds}(\text{course\_assigned}, T), \text{holds}(\neg\text{professor}, T) \quad (1)$$

or who is not employed by the university

$$\leftarrow \text{holds}(\text{course\_assigned}, T), \text{holds}(\neg\text{employed}, T) \quad (2)$$

Three additional actions can be used to achieve the final goal:

- hire (with precondition ¬employed and effects faculty and ¬professor) hires a person that is not registered in the personnel database as employed (this person may not be directly hired as a professor),
- promote (with precondition ¬professor and effect professor) which promotes a non-professor to the position of professor,
- register (with precondition ¬employed and effect employed) which registers a person as being employed in the database of the personnel department.



The final goal holds(course_assigned,T), holds(faculty,T) can be achieved by the partial-order plan (constructed by our algorithm) presented in the Figure above. Double arrows denote protection intervals not_clipped(T, F, T), while simple arrows are plain ordering constraints. Note that state constraints induce implicit preconditions as well as ordering constraints and protection intervals. For example, the ordering constraints r < a_c and pr < a_c have been posted to prevent the activation of the state constraints (2) and (1) respectively.

2 Problems with Abductive Procedures

Partial-order planning can be viewed as abductive planning in the Event Calculus [7]. Considering the fact that there are many partial-order planners for *independent* fluents, and since adding state constraints to an Event Calculus action specification seems straight-forward (see (8) below), it might seem that it should be easy to develop a partial-order planner dealing with *dependent* fluents.[1] This impression is however misleading: integrity constraints represent a significant complication. Intuitively, fluent dependencies seriously complicate the detection of '*threats*'. As far as we know, there are no implemented sound and complete partial-order planners able to deal with *non-ground plans* (i.e. plans with existentially quantified variables) and *dependent fluents* exist.[2] (Dependent fluents are fluents subject to integrity constraints. As usual, integrity constraints may have universally quantified variables.)

In the following we illustrate problems faced by existing abductive procedures, especially in our planning domain.

We start with the following (simplified[3]) *Abductive Logic Programming (ALP)* specification of the Event Calculus with dependent fluents:

P:
- holds(F,T) ← starts(F,T), T <T, not clipped(T ,F,T) (3)
- clipped(T ,F,T) ← starts(¬F,T), T <T , T <T (4)
- starts(F,0) ← initially(F) (5)
- starts(F,T) ← initiates(A,F), happens(A,T) (6)

I:
- ← happens(A,T), precondition(A,F), not holds(F,T) (7)
- ← holds(F ,T), …, holds(F_n,T) (8)

Abducibles A={happens}
Query: ?- holds(F ,T), …, holds(F_i,T_i), … T <T_j, …

where P are program clauses, while I are integrity constraints. A denotes actions, T time points and F fluent literals, i.e. (negated) atoms. starts(F,T) means that

---

1 Some implementations (including UCPOP) distinguish between *primitive* and *derived* fluents. (Primitive fluents are assumed *independent* and actions cannot have derived fluents as effects.) This is a simple form of *ramification* – the primitive fluents are still independent (state constraints are not allowed).

2 For example, the planner of [7] is unsound in the presence of state constraints (see the last example in section 6 of [7]). [1] has non-minimality problems (see below) and we haven't been able to use it for dependent fluents.

3 More complicated action languages can be easily incorporated.

fluent F becomes true at T, while starts(¬F,T) means that F is terminated at T. Direct effects are given by initiates, while preconditions are given by precondition. (8) represents a problem-dependent integrity constraint involving fluents F_1, …, F_n.

A purely backward abductive procedure (like *ACLP* [4] or *SLDNFA* [2]) typically faces problems in recognizing that an abducible selected for solving one subgoal also solves a second subgoal. If the second subgoal can be achieved in several ways, the first solution returned by the backward abductive procedure might be non-minimal, since it might *reachieve an already achieved goal*. For example, consider the following ALP P = {p ← a, q ← b, q ← a} with a, b abducibles and the query ?- p, q. For solving the first subgoal, p, the abductive procedure will abduce a. Then, when trying to solve q, it will abduce b, not realizing that q has already been solved by assuming a. (Both *ACLP* and *SLDNFA* have this problem, the latter even if a, b are declared strong abducibles.)

This problem could be solved by forward propagation of a to q: a ⇒ q. Now, when trying to solve q, the procedure would find that q has already been achieved.

Such situations occur very frequently in our planning domain, where an effect can be achieved by several actions. For example, if {initiates(a ,f), initiates(a ,f), initiates(a ,f)}, then the first answer to the query ?- holds(f ,t), holds(f ,t) will be non-minimal (both a and a will be applied).

There is a second situation in which existing abductive procedures might produce non-minimal solutions, namely the presence of *negative literals in negative goals*. As these are in fact disjunctions in disguise, existing abductive procedures will treat them by *splitting*, even in cases in which the negative goal is already achieved. For example, for the ALP P = {p ← a, q ← b, r ← b} with abducibles A = {a, b} and integrity constraints I = { ← not p, not q }, the query ?- r would first lead to abducing b for solving r, and then generate the negative goal ← not p, not q. Because b entails q, this goal is already solved, but a backward abductive procedure would not know this and would split the negative goal into the positive goal p and the negative goal ← not q (which will be reduced to q). Unfortunately, this would abduce a to achieve p, leading to a *non-minimal* solution.[4] (In other words, we are "repairing" an already repaired integrity constraint.) Again, propagating b forward to q would allow us to inactivate the negative goal ← not p, not q before splitting it.

This situation does not arise in planning with independent fluents (i.e. without state constraints). Several negated literals can appear in negative goals only due to the state constraints (8) (after unfolding holds to not clipped). The presence of state constraints therefore complicates partial-order planning to a significant extent. (The solution to the frame problem offered by partial-order planning is especially simple in the case of *independent* fluents.)

For example, consider the actions {initiates(a ,¬p), initiates(a ,¬q), initiates(a ,r), initiates(b ,p), initiates(b ,q), initiates(b ,s)}, the state constraint

← holds(p,T), holds(q,T), holds(s,T)

[4] *SLDNFA* has this problem too. Due to its syntactical restrictions, *ACLP* cannot even deal directly with this example.

and the goal ?- holds(r,t_n),…. Further assume that actions a , b , b , b solve all positive goals and have already been placed in the plan: happens(a ,t), happens(b ,t), happens(b ,t), happens(b ,t), together with the ordering constraints t <t , t <t , t <t, t<t . Now, the integrity constraint will be successively unfolded to the negative goals:

← happens(b ,T), not clipped(T ,p,T), T <T,
  happens(b ,T), not clipped(T ,q,T), T <T,
  happens(b ,T), not clipped(T ,s,T), T <T.
← not clipped(t ,p,T), not clipped(t ,q,T), not clipped(t ,s,T),
  T>max(t ,t ,t).

(the last step corresponds to the resolution with the abducibles happens(b ,t)).
The last negative goal is already solved because the presence of a in the plan ensures that clipped(t ,q,t) holds. Solving the negative goal by splitting, for example by generating the positive subgoal ?- clipped(t ,p,T), T>max(t ,t ,t) would place the additional unnecessary action a_1 in the plan.

A careful analysis of existing abductive procedures shows that both problems mentioned above involve reachieving an already achieved goal and are related to the treatment of implicit disjunctions by splitting.

We argue that in both cases we should avoid splitting disjunctions when these are already achieved. This will not *guarantee* the minimality of the *first* solution, but it will at least avoid reachieving already achieved goals. Of course, the minimal solution is in the search space, but in general we cannot guarantee obtaining it in polynomial time. More precisely, the problem of finding a (locally) minimal[5] explanation in an abductive problem with integrity constraints is *NP*-complete, even in the propositional case (Theorem 4.5 of [10]).

For example, considering the same ALP program P = {p ← a, q ← b, r ← b} as above, the query ?- q, p will lead in our framework to the non-minimal abductive explanation b,a. Note however, that the minimal solution a is in the search space and will be found upon backtracking.

3 Propagating Abductive Changes

We have seen that the planning problem can be formulated as an abductive problem. The main efficiency problem faced by all implemented abductive procedures is avoiding testing *all* integrity constraints after each abductive change. Since the integrity constraints have been tested before the change, we should retest only the ones that are influenced by the change in some abducible. For example, for achieving this, *ACLP* [4] requires each integrity constraint (IC) to contain at least an abducible predicate. The current implementation also requires each non-abducible positive condition in an IC not to depend on abducible predicates. If it does, as it is usually the case, the user would have to unfold the predicate in the IC with its definition until its dependence on the abducibles is made explicit. These strong requirements are needed so that there are only *direct* influences of changing abducibles on ICs. If this limitation is removed, then we need to be able

[5] *parsimonious* in the terminology of [10].

to determine which predicates are (indirectly) influenced by a change in an abducible. This can be achieved by *forward propagation* of the abductive changes from abducibles to other predicates occurring in ICs.

In the following we propose a mixed abductive procedure combining *backward* goal reduction rules with *forward* propagation rules for the abductive changes. In this problem solving strategy, the goals are reduced backwards to abducibles and constraints, which are then propagated forward ("saturated" to a complete solution). The role of the forward propagation rules is not only to detect inconsistencies, but also to *repair* any *potential* inconsistencies (by adding new abducibles and/or constraints). Instead of retesting all ICs after each modification, we propagate the change through the ICs and suggest repairs that ensure that the ICs are not violated.

3.1 Constraint Handling Rules

Constraint Handling Rules (CHRs) [3] represent a flexible approach to developing user-defined constraint solvers in a declarative language. As opposed to typical constraint solvers, which are black boxes, CHRs represent a 'no-box' approach to CLP. CHR propagation rules are ideal candidates for implementing the rules for forward propagation of abductive changes.

CHRs can be either *simplification* or *propagation* rules.

A *simplification* rule Head $\Leftrightarrow$ Guard | Body replaces the head constraints by the body provided the guard is true (the Head can contain multiple CHR constraint atoms).

Propagation rules Head $\Rightarrow$ Guard | Body add the body constraints to the constraint store without deleting the head constraints (provided the guard is true). A third, hybrid type of rules, *simpagation rules* Head \ Head $\Leftrightarrow$ Guard | Body replace Head by Body (while preserving Head) if Guard is true. (Guards are optional in all types of rules.)

4 Transforming ALPs into CHRs

In the following, we present a general transformation from an Abductive Logic Program (ALP) into a set of Constraint Handling Rules (CHRs), which function as an abductive procedure for the given ALP. We also illustrate the transformation on the example of partial-order planning with dependent fluents.

We start from an ALP $\langle P,A,I\rangle$. In the case of partial-order planning, we will use the ALP (3)-(8) from Section 2. Our goal is to replace the integrity constraints (ICs) (which would naively have to be retested after each abductive step) by forward rules for propagating abductive changes. This is useful both for detecting inconsistencies and for suggesting repairs. However, not every predicate in an IC can be the target of forward propagation - such predicates would have to be unfolded (backwards) until "forward" predicates are reached.

Since the specific problem may require certain predicates or rules to be "backward" -- even if they *could* in principle be "forward" -- we allow them to be *explicitly declared as "backward"*.[6] (This is obviously a problem-dependent specification.)

The transformation rules below will automatically determine the status (forward/backward) of the predicates and rules that are not explicitly declared as "backward".

Note that the transformation of ALPs into CHRs presented below is completely general (it works for any ALP program, not just for our planning domain).

1 First, we determine which rules and predicates can be treated by forward propagation (or, in short, are *"forward"*). (Rules and predicates which are not "forward" are called *"backward"* and will have to be treated by unfolding.)

- *Abducibles* (like happens in our planning domain) are automatically "forward". The occurrences of such abducibles p in rule bodies will be replaced by constraints $\mathcal{C}$p. (Intuitively, $\mathcal{C}$p denotes the "open" part of predicate p. Technically, $\mathcal{C}$p is used to trigger the forward propagation rules for p -- see (*) below).
- An *ALP rule* can be "forward" only if all its body literals are "forward". In particular, rules with negative literals in the body cannot be used as forward rules (for ex. rule (3)).
- Predicates which appear in the head of at least one "forward" ALP rule are themselves "forward".

2 Then, we replace the ICs by forward propagation rules. For each IC, we unfold the positive literals - in all possible ways *with their "backward" rules only* - until we are left with positive "forward" literals, negative literals or constraints:

$$\leftarrow p, \ldots, p_n, \text{not } q, \ldots, \text{not } q, c, \ldots, c$$

Such an unfolded IC is replaced by the forward propagation rule

$$\mathcal{C}p, \ldots, \mathcal{C}p_n \Rightarrow \sim c ; \ldots ; \sim c ; c, (q ; \ldots ; q) \qquad (*)$$

where $\sim c$ is the complement of the constraint c and c is the conjunction c , …, c (we apply a sort of semantic splitting w.r.t. the constraints).

This forward rule exactly captures the functioning of the abductive procedure, which waits for p , …, p_n and only then treats the remaining body by splitting. Note that the *subgoal* q_j is propagated (rather than the *constraint* $\mathcal{C}q_j$). Solving the subgoal q_j amounts to constructively ensuring that the IC is not violated (this functions as a *constructive "repair"* of a *potential* IC violation).

3 Finally, we replace negative literals not p in bodies of "backward" rules by constraints $\mathcal{C}$not_p (whose role will be to protect against any potential inconsistencies with some p).[7] For each such negative literal we add the IC $\leftarrow$ not_p, p, which will be treated as in step (2) above.

---

[6] For example, in our planning domain, we may wish to avoid propagating the initial state forward, especially if we are dealing with a database having a huge number of records.

[7] Unlike "normal" abducibles which are implicitly "minimized", abducibles of the form $\mathcal{C}$not_p are subject to a maximization policy. Thus, we cannot expect *all instances* $\mathcal{C}$not_p to be ex-

In our **planning domain**, rule (3) is backward because it has a negative literal (not clipped) in the body.

Rules with constraints[8] in the body could be used as forward rules, but they would propagate disjunctions (treated by splitting), which should be avoided. (4) will therefore be treated backward because of the constraints '<' in its body.

(5) will be treated backward because propagating the initial state of a database for example (having a huge number of records) is infeasible in practice.

(6) can be treated as forward, so starts will be a "forward" predicate: [9]

$$\mathbf{C}\text{happens}(A,T), \text{initiates}(A,F) \Rightarrow \mathbf{C}\text{starts}(F,T) \quad (9)$$

Similarly, the IC (7) induces the forward rule

$$\mathbf{C}\text{happens}(A,T), \text{precondition}(A,F) \Rightarrow \text{holds}(F,T) \quad (10)$$

(The action description predicates precondition and initiates are "static constraints", i.e. constraints that are given at the beginning of the problem solving process and which do not change.)

Finally, the ICs (8) related to state constraints are unfolded (since holds is a backward predicate) to

$$\leftarrow \text{starts}(F,T), \text{not clipped}(T,F,T), T<T, \ldots, \text{starts}(F_n,T_n), \text{not clipped}(T_n,F_n,T), T_n<T.$$

which induce the forward rules

$$\mathbf{C}\text{starts}(F,T), \ldots, \mathbf{C}\text{starts}(F_n,T_n) \Rightarrow T>\max(T,\ldots,T_n), (\text{clipped}(T,F,T); \ldots; \text{clipped}(T_n,F_n,T)). \quad (11)$$

(The first disjunct, $T<\max(T,\ldots,T_n)$, of the consequent could be dropped since it mentions the free variable T.)

Since starts also has a backward rule (5), the IC (8) also unfolds to

$$\leftarrow \ldots, \text{initially}(F), \text{not clipped}(0,F,T), \ldots, \ldots, \text{starts}(F_i,T_i), \text{not clipped}(T_i,F_i,T), T_i<T, \ldots$$

The induced propagation rule is

$$\ldots, \mathbf{C}\text{starts}(F_i,T_i), \ldots \Rightarrow \ldots; \sim\text{initially}(F); \ldots; \text{initially}(F), \ldots, \text{initially}(F), T>\max(T_i,\ldots), [\ldots; \text{clipped}(0,F,T); \ldots; \ldots; \text{clipped}(T_i,F_i,T); \ldots] \quad (12)$$

Finally, the IC

$$\leftarrow \text{not\_clipped}(T,F,T), \text{clipped}(T,F,T), F=F$$

is unfolded with (4) to

$$\leftarrow \text{not\_clipped}(T,F,T), \text{starts}(\neg F,T), T<T, T<T, F=F$$

and, since starts still has a "backward" rule (5), also to

$$\leftarrow \text{not\_clipped}(T,F,T), \text{initially}(F), T<0, 0<T, F=F$$

---

plicitly propagated and we should therefore avoid having to forward propagate $\mathbf{C}$not_p. not_p will thus be a backward predicate, used just to avoid violations of the IC □ not_p, p.

8 Here, by constraints we mean predicates for which the Closed World Assumption does not hold. For example, the absence of T1<T2 from the constraint store does not entail T1□T2.

9 Having both backward and forward versions of rule (6) does not lead to redundancies or loops. The role of the backward rule is to reduce a goal formulated in terms of starts(F,T) to assuming the abducible $\mathbf{C}$happens(A,T) for an action A that initiates F. Then the forward rule propagates *all* other effects of A (not just the one that triggered the action application).

The latter IC can be dropped since $T<0$ is inconsistent with the general constraint $0<T$. The forward propagation rule induced by the first IC is

$$\mathbf{C}\text{not\_clipped}(T,F,T), \mathbf{C}\text{starts}(\neg F,T) \Rightarrow F \neq F \;;\; F=F, (T>T \;;\; T>T) \quad (13)$$

We have thus obtained the following set of rules

Backward goal reduction rules

$$\text{holds}(F,T) \Leftrightarrow \text{starts}(F,T), T<T, \mathbf{C}\text{not\_clipped}(T,F,T) \quad (14)$$

$$\text{starts}(F,T) \Leftrightarrow \text{initially}(F), T=0 \;;\; \text{initiates}(A,F), \mathbf{C}\text{happens}(A,T) \quad (15)$$

$$\text{clipped}(T,F,T) \Leftrightarrow \text{starts}(\neg F,T), T<T, T<T \quad (16)$$

Forward propagation rules: (9)-(12).

We also have a general rule for all constraint predicates p:

$$\mathbf{C}p(X) \setminus p(X) \Leftrightarrow X=X \;;\; X \neq X, p(X) \quad (17)$$

which is given a higher priority than the other rules and which tries to solve a goal $p(X)$ by reusing an already existing constraint $\mathbf{C}p(X)$ (propagated earlier by a forward rule). This rule also leaves the alternative of constructively achieving $p(X)$ open.

4.1 An Improved CHR Implementation

While the above approach avoids reachieving already solved *positive* goals, it doesn't avoid splitting when dealing with negative literals in *negative* goals (in our case 'not clipped' in negative goals originating from state constraints). An improved implementation would have to explicitly represent the disjunctive goals (involving clipped) before actually splitting them.

We shall represent partially activated state constraints as $\text{ic}(\text{Head} \leftarrow \text{Body})$ (the initial state constraints have the form $\text{ic}(\text{fail} \leftarrow \text{Body})$). Like in rule (11), such integrity constraints are (partially) activated by $\mathbf{C}\text{starts}(F,T)$ constraints:

$$\mathbf{C}\text{starts}(F,T), \text{ic}(\text{Head} \leftarrow F, \text{Body}) \Rightarrow F =_{\square} F, \text{ic}(\text{Head} \;;\; \neg F \mid T \leftarrow \text{Body}) \;;\; \sim(F =_{\square} F) \quad (18)$$

where $\neg F \mid T$ denotes the fact that the IC has been activated by a fluent F becoming true at time point T ($F =_{\square} F_2$ is defined in Section 5.2 below).

The following rule inactivates an IC if a pair of its activating starts literals is clipped:

$$\mathbf{C}\text{starts}(\neg F,T), T<T, T<T_i \setminus \text{ic}(\text{Head};\neg F \mid T;\neg F_i \mid T_i \leftarrow \text{Body}) \Leftrightarrow F=F \;;\; \sim(F=F), \text{ic}(\text{Head};\neg F \mid T;\neg F_i \mid T_i \leftarrow \text{Body}) \quad (19)$$

(Note that this rule has the form $\mathbf{C}\text{clipped} \setminus \text{ic} \Leftrightarrow \text{true}$, or even

$\mathbf{C}\text{clipped} \setminus (\text{clipped} \;;\; \ldots \;;\; \text{clipped}) \Leftrightarrow \text{true}$.)

Finally, if an IC has been completely activated (without being inactivated by the previous rule, which has higher priority), then we should clip at least a pair of starts literals that activated it:

$$\text{ic}(\text{Head} \leftarrow \text{true}) \Rightarrow \text{ic\_clip}(\text{Head}) \quad (20)$$

Note that it is sufficient to clip a pair $(\neg F \mid T, \neg F_i \mid T_i)$ such that T has no known antecedent (w.r.t. the temporal order) in the set of activators, while T_i has no known successor:

```
ic_clip(Head) :-
    lower_bound(Head, ¬F | T), upper_bound(Head, ¬Fi | Ti), clipped(T, F, Ti).
```

We also have to deal with the possibility of ICs being activated by the initial state (while avoiding the forward propagation of the initial state):

$$\text{ic}(\text{Head} \leftarrow \text{Body}) \Rightarrow \textit{forall}\ \text{Body} = \text{F}, \ldots, \text{F}\ \textit{such that}\ \text{initially}(\text{Body}) \quad (21)$$
$$\text{ic}(\neg\text{F} \mid 0 ; \ldots ; \neg\text{F} \mid 0 ; \text{Head} \leftarrow \text{true}).$$

The ICs propagated by these rules can of course be inactivated by the previous inactivation rule (19). (Note that in the improved approach rules (18)-(21) replace rules (11) and (12).)

The above rules have been directly implemented in the ECLiPSe as well as SICStus CHR environments. We have run tests comparing a simple partial order planner for independent fluents (similar to UCPOP and SNLP) with the planner described above and noticed no overheads (due to the treatment of dependent fluents) on planning problems with *independent* fluents. Since there are no other planners dealing with non-ground plans and dependent fluents, no standard benchmarks are currently available. However, we have successfully tested the planner on query planning problems in system integration, where depedent fluents occur naturally (as briefly described in the Introduction).

5 A General Abductive Procedure

In order to better clarify the relationship of our approach to existing abductive procedures, we present in the following a *general* abductive procedure that tackles the above-mentioned non-minimality problems by allowing a limited form of forward reasoning in addition to backward goal-directed reasoning. The procedure doesn't aim at improving the implementation from Section 4.1, its main role being to generalise the approach from the previous Sections. The transformation algorithm from Section 4, which *compiles* an ALP to CHRs is replaced by a general abductive algorithm that *interprets* the ALP directly. Of course, an *interpreter* is slower than a *compiled* procedure. However, besides providing a clarification of our approach, the general abductive procedure can also be used as an intermediate step for proving the soundness and completeness of our partial-order planning algorithm for dependent fluents. (The implementation from Section 4.1 above is more efficient due to its direct encoding in CHR, as opposed to using CHR just for *interpreting* positive and negative goals, as below. A number of problem and domain dependent decisions, such as declaring certain predicates to be "backward", also influence the efficiency of the implementation from Section 4.1. Let us stress the fact that these decisions are entirely domain and problem dependent and that they are not a drawback of our general mechanism.)

5.1 Open Predicates

In Logic Programming, normal predicates are *closed*: their definition is assumed to be complete (Clark completion). On the other hand, abducibles in Abductive Logic Programming (ALP) are *completely open*, i.e. they can have any extension consistent with the integrity constraints. To formally deal with forward propaga-

tion rules in an abductive framework, we need to allow a generalization of abducibles, namely (partially) *open predicates*.

Unlike abducibles, open predicates can have definitions $p \leftarrow Body$, but these are not considered to be complete, since during the problem solving process we can add (by forward propagation) new abductive instances of p to these definitions. The definition of an open predicate therefore only partially constrains its extension.

In our CHR embedding of the abductive procedure we shall use two types of constraints, p and $\boldsymbol{C}p$, for each open predicate p. While $\boldsymbol{C}p$ represent facts explicitly propagated (abduced), p refers to the *current closure* of the predicate p (i.e. the explicit definition of p *together* with the explicitly abduced literals $\boldsymbol{C}p$). Thus, informally we have $p = def(p) \vee \boldsymbol{C}p$.

While propagating $\boldsymbol{C}p$ amounts to simply assuming p to hold (abduction), propagating p amounts to trying to prove p either by using its definition def(p), or by reusing an already abduced fact $\boldsymbol{C}p$.[10] This distinction ensures that our CHR embedding conforms to the usual '*propertyhood view*' on integrity constraints:

Definition $M(\Delta)$ is a *generalized stable model* of the abductive logic program $\langle P,A,I\rangle$ for the abductive explanation $\Delta \subseteq A$ iff

(1) $M(\Delta)$ is a stable model of $P \cup \Delta$, and (2) $M(\Delta) \models I$.

The distinction between propagating $\boldsymbol{C}p$ and p respectively can be seen best in an example. When an action is applied, $\boldsymbol{C}happens(A,T)$, we have to propagate its effects Eff as well as its preconditions Pre. But while propagating the effects simply involves the propagation of the *constraint* $\boldsymbol{C}starts(Eff,T)$, propagating the preconditions should entail posting the *goal* holds(Pre,T), which amounts to trying to achieve Pre either by using its definition (and thus applying another action having Pre as an effect), or by reusing an already achieved fact.

The use of *open predicates* allows mixing *forward propagation* of abduced predicates $\boldsymbol{C}p$ with *backward reasoning* using the closures p. Forward propagation can be implemented using CHR propagation rules, while backward reasoning involves unfolding predicates with their definitions. The definition def(p, Body) of a predicate p is obtained by Clark completion of its 'if' definitions. For each such predicate we will have an *unfolding rule* (a CHR simplification rule) $p \Leftrightarrow def(p, Body) \mid Body$, but also a CHR simpagation rule[12] for matching a goal p with an existing abduced fact $\boldsymbol{C}p$:

$$\boldsymbol{C}p(X) \setminus p(X) \Leftrightarrow X = X \ ; \ X \neq X \mid p(X).$$

This rule should be given a higher priority than the unfolding rule in order to avoid reachieving an already achieved goal. Combined with the forward propagation mechanism for $\boldsymbol{C}p$, it deals with the first non-minimality problem men-

---

[10] Abducibles (i.e. completely open predicates) p have no definition and are thus referred to as $\boldsymbol{C}p$.

[11] This is done by the CHR rule (10) which corresponds to the ALP *integrity constraint* (7). Note that while program clauses propagate $\boldsymbol{C}p$ constraints, integrity constraints propagate goals p, in line with Lin and Reiter's observation [6] that state (integrity) constraints are usually intimately tied with the qualification problem.

[12] The rule is more complicated in practice, due to implementation details.

tioned in Section 2. Note that, for completeness, we are leaving open the possibility of achieving p(X) using its definition or reusing other abduced facts.

Our treatment of open predicates p = def(p) $\vee$ $\mathcal{C}$p is slightly different than the usual method [8] of dealing with (partially) open predicates p by introducing a new predicate name p' (similar to our $\mathcal{C}$p) and adding the clause p $\leftarrow$ p' to the definition of p:

{p $\leftarrow$ Def, p $\leftarrow$ p'}. (**)

The difference is that whenever referring to p we are implicitly trying to prove p, either by using its definition def(p) or by reusing an already abduced fact $\mathcal{C}$p, *but without allowing such a* $\mathcal{C}$p *to be abduced in order to prove* p (whereas in (**) treating p $\leftarrow$ p' as a *program clause*[13] would allow p' to be abduced when trying to prove p). This is crucial for ensuring a correct distinction[14] between goals p and abducibles $\mathcal{C}$p mentioned above (otherwise we would treat the propagation of action preconditions incorrectly). Without making this distinction, *we wouldn't even be able to refer to the current closure of* p.

5.2 The Abductive Procedure

The abductive procedure for open predicates given below is written using CHR rules.[15] We assume that conjunction ',' and disjunction ';' in positive goals are dealt with implicitly (disjunction being treated by splitting). Integrity constraints $\leftarrow$G are represented as negative goals not(G).[16] The order of rules does matter: the first rule matching a newly introduced constraint will be activated. If it is a simplification rule, the subsequent rules will not get the chance to be executed.

In the rules below, we let p denote a predicate (possibly with variables). Multiple occurrences of p in a rule involves the unification of the corresponding literals. We also write p(X) (with X a tuple of variables) whenever we want to make the variables of p explicit.

The abductive procedure presented above is change-oriented, since a change in the abducible $\mathcal{C}$p will trigger in rule [NEG-ABD] a matching negative goal, as in other abductive procedures. The main difference lies however in that $\mathcal{C}$p can be *any* open predicate, not just a completely open one. Such open predicates can be targets of forward propagation rules: Body $\Rightarrow$ $\mathcal{C}$p. If these propagation rules represent the forward direction of some program rules p $\leftarrow$ Body, then these program rules may be excluded when unfolding negative goals in [NEG-UNF]. There, def_(p, Body) returns the backward definitions of p, i.e. the Clark completion of the program rules for which no forward propagation rules have been written. For predicates p with no backward definition (for example for abducibles), def_(p, Body) returns Body = fail and the rule [NEG-UNF] propagates not(fail) i.e. true.

---

13 In fact, p □ p' should be treated as an integrity constraint and not as a program clause.

14 This distinction is essential only for *partially* open predicates and not for completely open predicates (abducibles).

15 For lack of space, we omit the treatment of built-in constraints.

16 'not' is here a CHR constraint and should not be confused with the negation as failure operator used in logic programming.

Positive Goals

- $\mathbf{C}p(X) \setminus p(X) \Leftrightarrow X = X \;;\; \sim(X = X), p(X).$
- $p \Leftrightarrow def(p, Body) \mid Body.$

Negative Goals

- $not(true) \Leftrightarrow fail.$
  $not(fail) \Leftrightarrow true.$
- $not(p, G) \Rightarrow def\_(p,Body) \mid not(Body, G).$
- $\mathbf{C}p(X) \setminus not(p(X), G) \Leftrightarrow$
  $not(X =_{\square} X, G), not(p(X), \sim(X =_{\square} X), G)$
- $not((p ; p), G) \Leftrightarrow not(p, G), not(p, G).$
- $\mathbf{C}p(X) \setminus not(G, not\ p(X), G) \Leftrightarrow no\ \forall vars(X) \mid$
  $X = X \;;\; \sim(X = X), not(G, not\ p(X), G)$
- $not(not\ p(X), G) \Leftrightarrow no\ \forall vars(X) \mid p(X) \;;\; not(p(X)), not(G).$

The main improvements of this abductive procedure consist in solving the problems of reachieving already achieved goals (mentioned in Section 2):

- in the case of positive goals by forward propagation
- in the case of negative literals in negative goals by inactivating the negative goals (using [NEG-INACT], whenever possible) before splitting them (using [NEG-SPLIT]). This also relies on forward propagation.

Negative goals can contain both universal ($\forall$) and existential ($\exists$) variables (the latter correspond to anonymous constants occurring in the constraint store). For variable tuples X and X we denote by $X =_{\square} X$ the set of equations obtained after eliminating (unifying away) the $\forall$ variables.[17] The $no\ \forall vars(X)$ condition in the guard of [NEG-SPLIT] succeeds whenever the variable tuple X contains no universal variables. Its role is to avoid floundering. $not(p(X))$ in the second disjunct of the consequent of [NEG-SPLIT] implements a form of semantic splitting.

We have assumed conjunctions in the above algorithm to be ordered by a *selection function*. The selection function in a negative goal would typically prefer completely open predicates to other predicates and leave negative literals at the end. To avoid floundering, it would also try to choose only negative literals with no universal variables.

Our abductive procedure can be easily proved to be sound and complete. (The formal proof - which cannot be given here for lack of space - extends the standard proof for *SLDNFA* [2].)

For solving the planning problem with our general abductive procedure, we can simply use the general abductive logic program given by (3)-(8) together with the forward propagation rule (9) for starts. (Thus the rule (6) is "forward", the other ones, namely (3)-(5), being labeled "backward".)

---

[17] For example, if $X = [Y,Z,Z]$, $X_2 = [A,B,C]$ with Y,Z □ variables and A,B,C □ variables, $X =_{\square} X_2$ is the set of equations $\{B=C\}$ obtained after getting rid of Y and Z by $Y=A$, $Z=B$. We consider both cases $X =_{\square} X_2$ and $\sim(X =_{\square} X_2)$ in order to leave open the possibility that B□C.

6 Concluding Remarks

Several related works, such as [9], use forward propagation for incrementally checking the consistency of deductive databases. Although they use forward propagation, the Kowalski-Sadri [10] and related algorithms are not appropriate for our purposes, since they only check the consistency of an update. The intermediate forwardly propagated facts are neither retained, nor reused after the check. These algorithms are therefore of no help to us for avoiding reachieving in a different way an already achieved goal.

On the other hand, the *Suspended Logic Programs* (SLPs - similar with Fung's IFF procedure) of [5], also allow a combination of backward goal-oriented reasoning with forward propagation of necessary properties. SLPs are very similar to CHRs in that insufficiently instantiated goals (that do not *match* any head of their iff definitions) are suspended. Thus, suspension is used as a mechanism for avoiding the combinatorial explosions that would be entailed by unfolding insufficiently instantiated goals. Instead of unfolding them, forward rules[18] are used to propagate the properties of such suspended goals in the hope of discovering any potential inconsistencies or for further instantiating the goal variables and thus allowing their unfolding.

Unfortunately, the propagated properties have to be unfolded as well, which may lead to a blow-up of the computation, unless special care is given to suspension control.[19] In our terminology, such SLP propagation rules propagate goals of the form p (which are subject to unfolding, leading to potential blow-ups), while we only propagate forward constraints of the form $\mathbf{C}$p (which are *not* subject to unfolding). Thus, SLPs represent a more general architecture (very much like CHRs), while we are developing a more specific abductive procedure, being more concerned with dealing with the non-minimalities in existing abductive procedures.

As far as we know, our planner is the first sound and complete partial-order planner able to deal with dependent fluents and non-ground plans. The prototype CHR implementation has been successfully tested as query planner of the mediator in the framework of system integration.

References

[1] Denecker M., Missiaen L., Bruynooghe M. Temporal reasoning with abductive event calculus, ECAI-92, 384-388.

[2] Denecker M., DeSchreye D. SLDNFA: an abductive procedure for abductive logic programs, J.Logic Programming 34(2),111-167, 1998.

[3] Fruewirth T. Constraint Handling Rules, in Podelski A. (ed) Constraint Programming: Basic and Trends, LNCS 910, 90-107, 1995.

[4] Kakas A., Michael A., Mourlas C. ACLP: a case for non-monotonic reasoning, Proc. NM'98.

18 Similar to CHR propagation rules.

19 In fact, Wetzel started studying the use of deletion rules [11] for alleviating the blow-up of propagated properties.

[5] Kowalski R., Toni F., Wetzel G. Executing suspended logic programs, Fundamenta Informaticae 34 (1998), 1-22.
[6] Lin F., Reiter R. State constraints revisited, JLC4(5), 655-678.
[7] Shanahan M. An abductive event calculus planner, JLP44 (2000)
[8] Kakas A., Kowalski R., Toni F. The role of abduction in logic programming, Handbook of logic in AI and LP 5, OUP1998, 235-324.
[9] Kowalski R., Sadri F., Soper P. Integrity checking in deductive databases, VLDB'97.
[10] Bylander T., Allemang D., Tanner M.C., Josephson J.R. The computational complexity of abduction, AIJ 49(1-3), pp. 25-60, 1991.
[11] Wetzel G. Using integrity constraints as deletion rules, Proc. DYNAMICS'97, 147-161, 1997.

Constraint-Based Optimization of Priority Schemes for Decoupled Path Planning Techniques

Maren Bennewitz[1], Wolfram Burgard[1], and Sebastian Thrun[2]

[1] Department of Computer Science, University of Freiburg, Freiburg, Germany,
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh PA, USA

Abstract. Coordinating the motion of multiple mobile robots is one of the fundamental problems in robotics. The predominant algorithms for coordinating teams of robots are decoupled and prioritized, thereby avoiding combinatorially hard planning problems typically faced by centralized approaches. While these methods are very efficient, they have two major drawbacks. First, they are incomplete, i.e. they sometimes fail to find a solution even if one exists, and second, the resulting solutions are often not optimal. In this paper we present a method for finding and optimizing priority schemes for such prioritized and decoupled planning techniques. Existing approaches apply a single priority scheme which makes them overly prone to failure in cases where valid solutions exist. By searching in the space of priorization schemes, our approach overcomes this limitation. It performs a randomized search with hill-climbing to find solutions and to minimize the overall path length. To focus the search, our algorithm is guided by constraints generated from the task specification. To illustrate the appropriateness of this approach, this paper discusses experimental results obtained with real robots and through systematic robot simulation. The experimental results illustrate the superior performance of our approach, both in terms of efficiency of robot motion and in the ability to find valid plans.

1 Introduction

Path planning is one of the fundamental problems in mobile robotics. As mentioned by Latombe [10], the capability of effectively planning its motions is "eminently necessary since, by definition, a robot accomplishes tasks by moving in the real world."

In this paper we consider the problem of motion planning for multiple mobile robots. In particular, we are interested in planning paths for multiple robots operating in a single, shared environment, where physical limitations impose restrictions among the paths of the various robots. In such multi-robot problems, undesirable situations include congestions or deadlocks, which may prevent robots from reaching their goal locations. Since the size of the joint state space of the robots grows exponentially in the number of robots, planning paths for teams of mobile robots is significantly harder than the path planning problem for single robot systems. Therefore, existing approaches for single robot systems cannot directly be transferred to multi-robot systems.

The approaches for multi-robot path planning can roughly be divided into two major categories [10]: centralized and decoupled. In the *centralized* approach [3,19] the configuration spaces of the individual robots are combined into one composite configuration space which is then searched for a path for the whole composite system. Because

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 78–93, 2001.

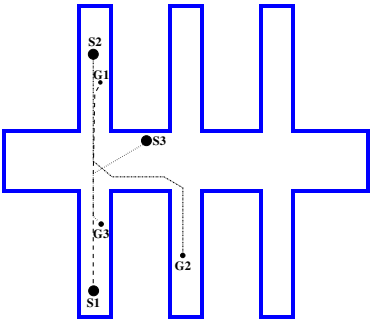


Fig. 1. Situation in which no solution can be found if robot 3 has higher priority than robot 1.

the size of the joint configuration space grows exponentially in the number of robots, this approach suffers intrinsic scaling limitations. The major alternative are *decoupled* approaches [7,17,13,5,21,1,8]. Decoupled approaches compute separate paths for the individual robots. Subsequently, they apply heuristics for resolving conflicts between different robots (e.g., two robots attempt to occupy the same location at the same time). To deal with the still enormous search space, it is common practice to assign *priorities* to the individual robots [7,5,21,1,8]. Planning and re-planning is performed in accordance with these priorities. Priority schemes provide an effective mechanism for resolving conflicts that is computationally extremely efficient.

However, the priority scheme has a strong influence on whether a solution can be found and on how long the resulting paths are. To illustrate this, let us consider two examples. Figure 1 shows a situation in which no solution can be found *if* robot 3 has a higher priority than robot 1. Since the path of robot 3 is planned without considering robot 1, it enters the corridor containing its target location (marked G3) before robot 1 has left this corridor. Since the corridors are too narrow to allow two robots to pass by, robot 3 blocks the way of robot 1 so that it cannot reach its target point G1. However, if we change the priorities and plan the trajectory of robot 1 before that of robot 3, then robot 3 considers the trajectory of robot 1 during path planning and thus will wait in the hallway until robot 1 has left the corridor. Another example is shown in Figure 2 (left). If we start with robot 1 then we have to choose a large detour for robot 2 (see Figure 2, center). This is because robot 1 blocks the corridor. However, if the path of robot 2 is planned first, then we can obtain a much more efficient solution (see Figure 2, right). These two examples illustrate that the priority scheme has a serious influence on whether a solution can be found and on how long the resulting paths are. Moreover, it suggests that no single prioritization scheme will be sufficient for all possible multi-robot motion problems.

In this paper, we present a technique that searches in the space of all priority schemes while solving hard multi-robot planning problems. Our approach performs a randomized hill-climbing search in the space of possible priority schemes. Since each change of a scheme requires the computation of the paths for many of the robots, it is important to focus the search. Our method achieves this by exploiting constraints between the different

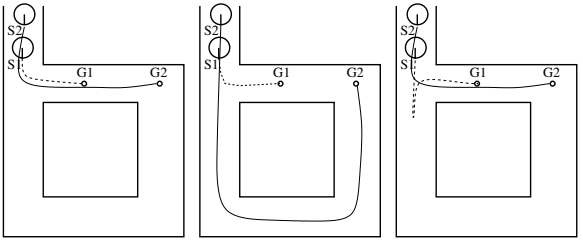


Fig. 2. Independently planned optimal paths for two robots (left), sub-optimal solution if robot 1 has higher priority (center), and solution resulting if the path for robot 2 is planned first (right) .

robots which are derived from the task specification. This has two serious advantages. First, it reduces the time required to find a solution, and second, it increases the number of problems for which a solution can be found in a given amount of time. Additionally, our algorithm is able to reduce the overall path length. It has anytime characteristics [22], which means that the quality of the solution depends on the available computation time; however, a solution may be available at any point in time.

Our approach has been successfully applied to physical mobile robots. These results are complemented by extensive simulations, to characterize the relation between the planning performance and various problem parameters. In all experiments, we found that our approach produces highly efficient motion plans even for very large teams of robots, for different environments, and using two different decoupled path planning techniques.

The paper is organized as follows. After discussing related work in the following section, we introduce two decoupled path planning techniques that will be used throughout this paper. Section 4 describes our algorithm for searching for priority schemes during planning. Finally, in Section 5, we present systematic experimental results illustrating the capabilities of our approach. The paper is concluded in Section 6.

2 Related Work

The problem of coordinating multiple mobile robots has received considerable attention in the robotics literature. As already mentioned above, the techniques for multi-robot path planning can roughly be divided into the *centralized* and the *decoupled* approaches [10].

Centralized methods consider the composite configuration space of all robots and search for a solution in the whole composite system. While these approaches (at least theoretically) are able to find the optimal solution to any planning problem for which a solution exists, their time complexity is exponential in the dimension of the composite configuration space. In practice, one is therefore forced to use heuristics for the exploration of the huge joint state space. Many methods use potential fields [2,3,20] to guide the search. These techniques apply different approaches to deal with the problem of local minima in the potential function. Other methods restrict the motions of the robots to reduce the size of the search space. For example, [9,19,11] only consider trajectories that lie on independent road-maps. The coordination is achieved by searching the Cartesian

product of the separate road-maps. Nevertheless, centralized approaches scale poorly to large numbers of robots.

Decoupled planners, in contrast, determine the paths of the individual robots independently and then employ different strategies to resolve possible conflicts. According to that, decoupled techniques are incomplete, i.e., they may fail to find a solution even if there is one. A popular decoupled approach is planning in the configuration time-space [7], which can be constructed for each robot given the positions and orientations of all other robots at every point in time. Techniques of this type assign priorities to the individual robots and compute the paths of the robots based on the order implied by these priorities. The method presented in [21] uses a fixed order and applies potential field techniques in the configuration time-space to avoid collisions. The approach described in [8] also uses a fixed priority scheme and chooses random detours for the robots with lower priority.

Another approach to decoupled planning is the path coordination method which was first introduced in [17]. The key idea of this method is to keep the robots on their individual paths and let the robots stop, move forward, or even move backward on their trajectories in order to avoid collisions (see also [6,4]). To reduce the complexity in the case of huge teams of robots, Leroy and colleagues [13] recently presented a technique to separate the overall coordination problem into sub-problems. Their approach, however, assumes that the overall problem can be divided into very small sub-problems. As various examples described below demonstrate, this assumption may not be justified in certain situations.

Unfortunately the problem of finding the optimal schedule is NP-hard for most of the decoupled approaches. To see, we notice that the NP-hard Job-Shop Scheduling problem with the goal to minimize maximum completion time [14,12] can be regarded as a special instance of the path coordination method. The decoupled and prioritized methods described above leave open how to assign the priorities to the individual robots. In the past, different techniques for selecting priorities have been used. For example, in [5] heuristic techniques are described that assign higher priority to robots which can move on a straight line from the starting point to their target location. In [1] all possible priority assignments are considered. Due to its (exponential) complexity this approach has only been applied to groups of up to three robots.

In this paper we present an approach to optimize priority schemes for arbitrary decoupled path planning methods. We perform a randomized hill-climbing search in the space of priority schemes. Thereby, we interleave the search for an optimal priority scheme with the planning of the paths of the robots. To guide the search, our algorithm exploits constraints between the robots that are extracted from the task description. As a result, our approach seriously reduces the time needed to find a solution to the path planning problem. Once a solution has been found, our algorithm is able to optimize the priority scheme in order to minimize the overall path length.

3 Prioritized A^*-Based Path Planning and Path Coordination

The basic algorithm to compute optimal paths for single robots, which will be used throughout this paper, is a variant of the popular A^* search procedure [16]. To represent

the environment of the robots we apply occupancy grids [15] which separate the environment into a grid of equally spaced cells and store in each cell $\langle x, y\rangle$ the probability $P(occ_{x,y})$ that it is occupied by a static object. In this section we also present the key ideas of decoupled prioritized path planning and describe how the A^* procedure can be utilized to plan the motions of teams of robots.

3.1 A^*-Based Path Planning

Our system applies the A^* procedure to compute the cost-optimal paths for the individual robots, in the remainder denoted as the independently planned optimal paths for the individual robots. A^* addresses the problem of finding a shortest path from an initial state to a goal state in a graph. To search efficiently, the A^* procedure takes into account the accumulated cost of reaching a certain location $\langle x, y\rangle$ from the starting position, and an estimate of the cost of reaching the target location $\langle x^*, y^*\rangle$ from $\langle x, y\rangle$. By doing so, A^* tends to focus its search in parts of the state space most relevant to the problem of finding a shortest path. This property, which makes A^* an efficient search algorithm, has given A^* an enormous popularity in the robotics community. However, A^* also requires a discrete search graph, whereas robot configuration spaces are continuous. In our case we assume that the environment is readily represented by a discrete occupancy grid map—which is common in the mobile robotics literature.

The cost for traversing a cell $\langle x, y\rangle$ is proportional to its occupancy probability $P(occ_{x,y})$. Furthermore, the estimated cost for reaching the target location is approximated by $c \cdot ||\langle x, y\rangle - \langle x^*, y^*\rangle||$ where $c > 0$ is chosen as the minimum occupancy probability $P(occ_{x,y})$ in the map and $||\langle x, y\rangle - \langle x^*, y^*\rangle||$ is the straight-line distance between $\langle x, y\rangle$ and $\langle x^*, y^*\rangle$. Since this heuristic is admissible (see [16]), A^* determines the cost-optimal path from the starting position to the target location.

3.2 Decoupled Path Planning for Teams of Robots

A^* can easily be extended to the problem of decoupled and prioritized path planning. Recall that in the multi-robot path planning problem, many robots simultaneously seek to traverse an environment. If the robots could move freely regardless of other robot's positions, the problem could easily be decoupled into many local path planning problem, in which each robot applied A^* to determine its optimal path. However, the impossibility for robots to occupy the same location at the same point in time introduces non-trivial restrictions that have to be incorporated into the individual robot paths.

A common approach is the following. In a first path planning step, each robot computes its optimal path using A^*, without any consideration of the paths of the other robots. Clearly, the resulting paths might not be admissible since they lead to collisions, if executed. Thus, in a second planning step, each robot checks for possible conflicts with all other robots. Conflicts between robots are then resolved by introducing a priority scheme. A priority scheme determines the order in which the paths for the robots are re-planned. The path of a robot is then planned in its configuration time-space computed based on the map of the environment and the paths of the robots with higher priority.

Please note, the A^* search can also be used to plan the motions of the robots in the configuration time-space. As in the standard approach described above, the cost

of traversing a location $\langle x, y \rangle$ at time t is determined by the occupancy probability $P(occ_{x,y})$. To incorporate the restrictions imposed by the paths of the other robots, however, we do not allow a robot to enter a cell that is occupied by a robot with higher priority at time t. In addition to the general A^*-based planning in the configuration time-space we consider a second and restricted version of this approach denoted as the path coordination technique [13]. It differs from the general approach in that it only explores a subset of the configuration time-space given by those states which lie on the initially optimal paths for the individual robots. The path coordination technique thus forces the robots to stay on their initial trajectories. The overall complexity of both approaches is $O(n \cdot m \cdot \log(m))$ where n is the number of robots and m is the maximum number of states expanded by A^* during planning in the configuration time-space (i.e. the maximum length of the OPEN-list). Due to the restriction during the search, the path coordination method is more efficient than the general A^* search. Its major disadvantage, however, lies in the fact that it fails more often.

As already discussed above, the introduction of a priority scheme for the decoupled path planning leads to serious reduction of the overall complexity. Whereas there are schemes leading to a viable solution with collision-free paths, it is easy to see that there are schemes for which no solution can be found. In addition to the fact, that the order in which the robots may plan their paths has a profound impact on the ability of finding a solution, even the quality of the solution depends heavily on the priority scheme. Examples of such situations were already discussed in the introduction to this paper. Unfortunately, the problem of finding the optimal priority scheme, is a non-trivial matter. More specifically, the NP-hard Job-Shop Scheduling problem with the goal to minimize maximum completion time [14,12] can be regarded as an instance of the path coordination method. Therefore, we have to be content with possibly sub-optimal planning orders.

4 Finding and Optimizing Solvable Priority Schemes

This section describes our approach to searching in the space of priority schemes during decoupled path planning. As the examples given in Figures 1 and 2 illustrate, the order in which the paths are planned has a significant influence on whether a solution can be found and on how long the resulting paths are. This raises the question of how to find a priority scheme for which the decoupled approach does not fail and how to find the order of the robots leading to the shortest paths.

4.1 The Randomized Search Technique

Our algorithm for finding eligible priority schemes is a randomized search technique, similar to those reported in [18]. More specifically, our approach performs a randomized and hill-climbing search in order to optimize the planning order for decoupled and prioritized path planning techniques. Our approach starts with an arbitrary initial priority scheme Π and randomly exchanges the priorities of two robots in this scheme. If the new order Π' results in a solution with shorter paths than the best one found so far, we continue with this new order. Since hill-climbing approaches like this frequently get

stuck in local minima, we perform random restarts with different initial orders of the robots. Thus, our approach interleaves the search for collision-free paths with the search for a solvable priority scheme.

4.2 Exploiting Constraints to Focus the Search

Whereas the plain randomized search technique produces good results, it has the major disadvantage that often a lot of iterations are necessary to come up with a solution. For example, we found that for ten robots in the environment shown in Figure 1 more than 20 iterations on average were necessary to find a solvable priority scheme. In this section we therefore present a technique to focus the search that tends to reduce the search time significantly. Our approach can be motivated through the situation depicted in Figure 1. In this situation, it impossible to find a path for robot 1 if the path of robot 3 is planned first, because the goal location of robot 3 lies on the optimal path for robot 1. The key idea of our approach is to introduce a constraint $p_i > p_j$ between the priorities of two robots i and j, whenever the goal position of robot j lies on the optimal path of robot i. In our example we thus obtain the constraint $p_1 > p_3$ between the robots 1 and 3. Additionally, we get the constraint $p_2 > p_1$, since the goal location of robot 1 lies too close to the trajectory of robot 2.

Although the satisfaction of the constraints by a certain priority scheme does not guarantee that valid paths can be found, orders satisfying the constraints more often have a solution than priority schemes violating constraints. Unfortunately, depending on the environment and the number of the robots, it is possible that there is no order satisfying all constraints. In such a case the constraints produce a cyclic dependency. The key idea of our approach is to initially reorder only those robots that are involved in such a cycle in the constraint graph. Thus, we separate all robots into two sets. The first group R_1 contains all robots that, according to the constraints, do not lie on a cycle and have a higher priority than the robot with highest priority which lies on a cycle. This set of robots is ordered according to the constraints and this order is not changed during the search. The second set, denoted as R_2 contains all other robots.

As an example, Figure 3 (left) shows a simulated situation with ten robots. Whereas the starting positions are marked by $S_0, \ldots, S_9$ the corresponding goal positions are marked by $G_0, \ldots, G_9$. The independently planned optimal trajectories are indicated by solid lines. Given these paths we obtain the constraints depicted in Figure 3 (right). According to the constraints, six robots belong to the group of robots whose order (at least in the beginning) remains unchanged during the search process. The robots in their order of priorities are 3, 6, 7, 2, 4, 9.

Initially, our algorithm only changes the order of the robots in the second group. After k iterations, we include all robots in the search for a priority scheme. In extensive experimental results we figured out that this approach leads to better results with respect to the overall path length, especially for large numbers of iterations. The complete algorithm is listed in Table 1.

If we apply this algorithm to the example shown in Figure 3 (left) under the constraints shown in Figure 4, the system quickly finds a solution. One typical result is the the order 0, 1, 5, and 8, for those robots that generate a cycle in the constraint graph.

Table 1. The algorithm to optimize priority schemes.

```
count := 0
FOR tries := 1 TO maxTries BEGIN
    IF count > k   // extensive search after k iterations
        select random order Π
    ELSE
        select order Π given fixed order for R1
        and random order for R2
    IF (tries = 1)
        Π* := Π
    FOR flips := 1 TO maxFlips BEGIN
        IF count > k // extensive search after k iterations
            choose random i, j with i<j
        ELSE
            choose random i, j with i<j and i,j∈ R2
        Π' := swap(i, j, Π)
        count := count+1
        IF moveCosts(Π') < moveCosts(Π)
            Π := Π'
    END FOR
    IF moveCosts(Π) < moveCosts(Π*)
        Π* := Π
END FOR
RETURN Π*
```

The corresponding collision-free paths for all robots are shown in Figure 3. This demonstrates, that the constraints drastically reduce the search space and still allow the system to quickly find solvable priority schemes.

5 Experimental Results

Our approach has been tested thoroughly on real robots and in extensive simulation runs. The two key questions addressed in our experiments were: (1) Solvability: Does our approach succeed more frequently in finding valid multi-robot paths than approaches with fixed prioritization? (2) Optimality: If our approach succeeds, does it generate more efficient plans? All experiments were carried out using different environments. To evaluate the general applicability, we applied our method to the two decoupled and prioritized path planning techniques described above. The current implementation is highly efficient. It requires less than 0.1 seconds on a 1000 MHz Pentium III to plan a collision-free path for one robot in all environments described below. The whole optimization for 10 robots with 10 restarts and 10 iterations per restart requires approximately one minute.

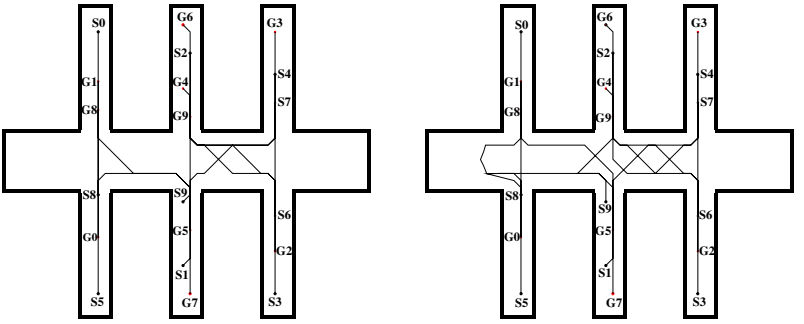


Fig. 3. Independently planned paths for ten robots (left) and the paths resulting after priority optimization (right).

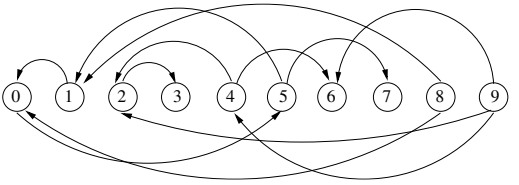


Fig. 4. Constraintgraph generated according to the paths shown in Figure 3 (left).

5.1 Simulation Experiments

To elucidate the scaling properties of our approach to larger number of robots, we performed extensive simulation experiments. In particular, we were interested in characterizing the dependence between the performance of our system on various components of our approach. In our experiments, we analyzed the number of planning problems that can be solved using our strategy, the speed-up obtained by exploiting the constraints, and the reduction of the overall path length. In all experiments, we found that our approach produces highly efficient motion plans even for very large teams of robots, for different environments, and regardless of the specific baseline path planning technique (e.g., A^*).

Solved Planning Problems. This first set of experiments was designed to characterize the effect of our search scheme on the overall number of failures. For each number of robots considered, we performed 100 experiments. In each experiment we randomly chose the starting and target locations of the robots. We applied four different strategies to find solvable priority schemes:

1. A single randomly chosen order for the robots.
2. A single order which satisfies the constraints for the robots in R_1 and consists of a randomly chosen order for the robots in R_2.
3. Unconstrained randomized search starting with a random order and without considering the constraints.

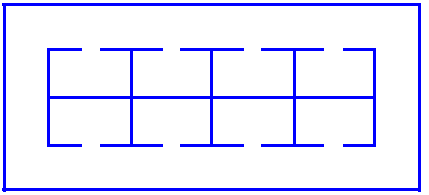


Fig. 5. Cyclic corridor environment used for the simulation runs .

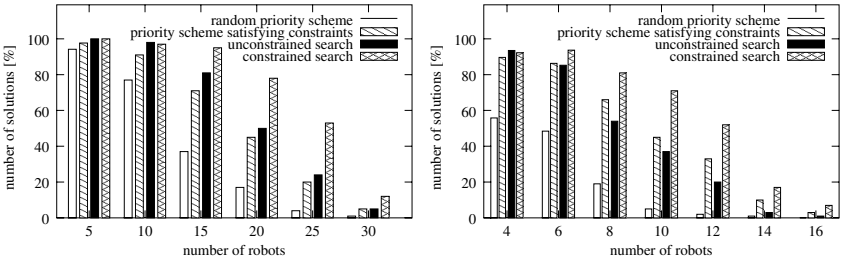


Fig. 6. Solved planning problems for different strategies using A^*-based planning in the configuration time-space in the cyclic corridor environment depicted in Figure 5 (left) and the corresponding results obtained in the noncyclic corridor environment shown in Figure 3 (right).

4. Constrained randomized search starting with an order computed in the same way as strategy 2).

All four strategies can be cast as special cases of our algorithm. In the first two strategies the corresponding values for `maxTries` and `maxFlips` are 1. For the first strategy the value of the threshold k is 0. The strategies 3 and 4 only differ in the value of the threshold k. Whereas the unconstrained search is obtained by setting $k = 0$, the constrained search corresponds to a value of $k = \infty$.

Please note that in this experiment we chose a small number of iterations for the last two strategies in order to assess the advantages of the constrained search under serious time constraints. Particularly, we chose a value of 3 for the parameters `maxFlips` and `maxTries`. Obviously, the larger the number of iterations, the higher is the probability that a solution can be found by an arbitrary randomized search. However, larger numbers of iterations drastically increase the computation time. For each technique, we performed A^*-based planning in the configuration time-space and counted the number of solved planning problems.

Figure 6 (left) summarizes the results we obtained for the cyclic corridor environment depicted in Figure 5. The horizontal axis represents the number of robots, and the vertical axis depicts the percentage of solved path planning problems. As this result illustrates, our constrained search technique succeeds more often than any of the alternative strategies. It is interesting to note that the second strategy, which exploits the constraints but considers only one scheme in each experiment, shows a similar performance than the unconstrained randomized search. To complement these results, we performed a similar

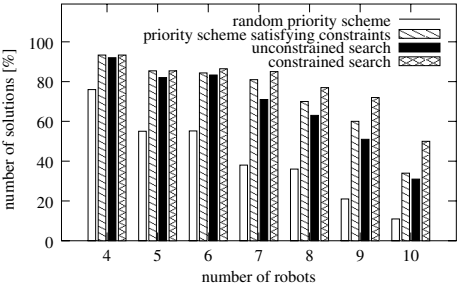


Fig. 7. Solved planning problems for all four strategies using the path coordination method in the noncyclic environment depicted in Figure 3 (left).

series of experiments for the noncyclic corridor environment depicted in Figure 3. The results are shown in Figure 6 (right). Again, our constrained-based search outperforms all other strategies. All these and the following results are significant on the 95% confidence level.

To investigate the performance using a different baseline path planning algorithm, we applied all four strategies using the path coordination method instead of plain A^*. We used a variant of the environment depicted in Figure 3 with five corridors on both sides. Since the path coordination method restricts the robots to stay on their independently planned optimal trajectories, the number of unsolvable problems is much higher compared to the A^*-based planning in the configuration time-space. As can be seen from Figure 7, our constrained search leads to a much higher success rate that actually increases with the number of robots involved.

Speed-up Obtained by Exploiting the Constraints. In this section, we are interested in one particular aspect of our approach, namely the ability to guide the search in the space of all priority schemes. More precisely, we pose the question how much the computation time necessary to find a solution can be reduced by constraining the search.

For the next set of experiments we increased the values of `maxFlips` and `maxTries` to 10 and evaluated in which iteration the first solution was found if the planning problem could be solved. Figure 8 (left) plots the results obtained for different number of robots in the cyclic corridor environment and Figure 8 (right) shows the same evaluation for the noncyclic environment. As can be seen, for both environments the unconstrained search needs significantly more iterations to generate a solution.

As these experiments suggest, the advantages of our constrained search is two-fold. On one hand, it requires fewer iterations than the unconstrained counter-part. On the other hand, it requires less computation, since the search is restricted to a subset of the robots, which reduces the number of paths that are generated in the search.

Influence on the Overall Path Length. The previous experiments investigated the number of cases in which a solution can be found, as a function of the algorithm used for path planning. In this section, we will be interested in plan efficiency, that is, the overall plan execution time.

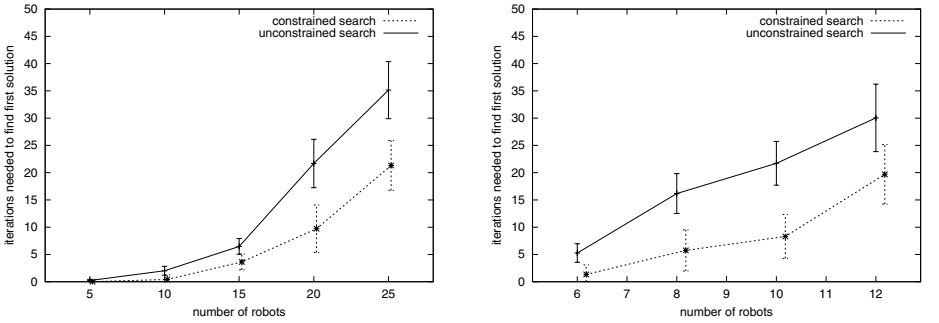


Fig. 8. Iteration in which the first solution was found if the planning problem could be solved for the cyclic corridor environment (left) and the corresponding results obtained int the noncyclic corridor environment (right).

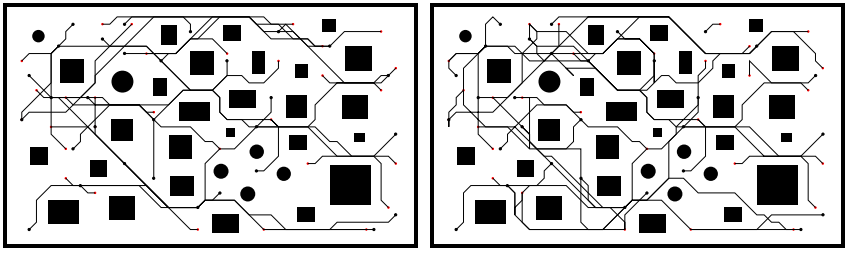


Fig. 9. Independently planned optimal paths for 30 robots (left) and the resulting paths after priority optimization (right).

To show that our optimization technique is not limited to typical corridor environments, Figure 9 (left) shows the independently planned optimal paths for a team of 30 robots in an unstructured environment. By optimizing these paths over 100 iterations, we obtain the solution illustrated in Figure 9 (right). Figure 10 (left) plots the evolution of the summed move costs of the best solution found so far over time. As can be seen from the figure, after 100 iterations the overall move costs are reduced by 15%.

The final experiment in this section is designed to analyze the performance of our algorithm with respect to the overall path length. Since our algorithm in the beginning only considers a restricted set of priority schemes, and after k iterations explores the whole set of priority schemes, we are especially interested in how long the resulting paths are compared to the unconstrained search. We performed over 100 experiments in the cyclic corridor-environment and determined the average overall move costs at every iteration. The corresponding graphs are shown in Figure 10 (right). This plot contains the average move costs for three different strategies at each iteration. The first data set was obtained for the constrained search which corresponds to $k = \infty$. Using this strategy we reorder only those robots which lie on a cycle in the constraint graph. The data for the unconstrained search was obtained using $k = 0$. In this case our algorithm chooses arbitrary priority schemes regardless of the constraints which were

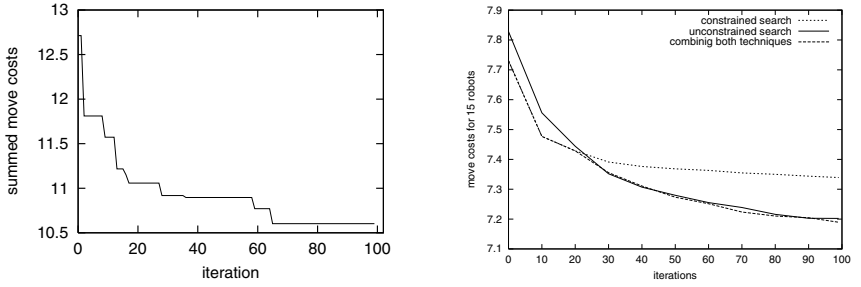


Fig. 10. Summed move costs plotted over time for the planning problem with 30 robots shown in Figure 9 (left) and summed move costs plotted over time averaged over 100 planning problems for 15 robots in the cyclic environment (right).

extracted given the task specification. Finally, the third function labeled "combining both techniques" corresponds to the results obtained with our algorithm given $k = 20$.

Since the constrained search, which is guided by our heuristics, focuses the search on the robots that pose the most serious restrictions to the other robots, it more quickly finds a solution and accordingly has more time to optimize it. Thus, in the beginning, the constrained search outperforms the unconstrained search. After 20 iterations, however, the situation completely changes. Because the unconstrained search can explore many more priority schemes, it more often finds better solutions than the constrained search. Thus, after 20 iterations, the unconstrained search leads to better results than the constrained search. As can be seen from the figure, our approach combines the advantages of both methods. In the beginning, it applies the constraints to focus the search and to quickly find a first solution which is optimized subsequently. After 20 iterations it considers arbitrary priority schemes so that the resulting path length is reduced as in the unconstrained search.

Accordingly, our randomized search that initially uses the constraints to focus the search for a viable solution and afterwards uses the unconstrained search to optimize this solution inherits the advantages of both techniques with respect to efficiency and the overall resulting path length.

5.2 An Example with Two Real Robots

Figure 11 (center) illustrates a typical application example carried out in our office environment with our robots Albert and Ludwig. The robots are shown in Figure 11 (left and right). In this example, we used the general A^* procedure in the configuration time-space for local path planning. While Albert starts at the right end of the corridor of our lab and has to move to left end, Ludwig has to traverse the corridor in the opposite direction. Notice that no path for Albert can be found if the path of Ludwig is planned first, since Albert cannot reach its target point if Ludwig stays on its optimal trajectory. Because of that, the system alters the order of the two robots. Given the optimal path for Albert, our system plans a path for Ludwig which first leads it into a doorway in order to let Albert pass by. The resulting trajectories are shown in Figure 11 (center). Notice that

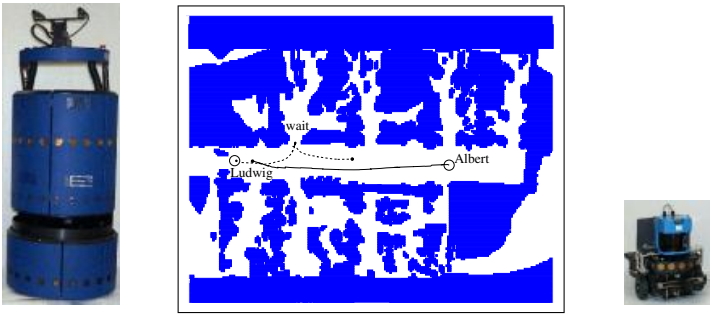


Fig. 11. The mobile robots Albert (left) and Ludwig (right) and a real world application of A^*-based planning in the configuration time-space where Ludwig moves away in order to let Albert pass by (center) .

at some point, the robot Ludwig waits to let the robot Albert pass by. In comparison, no solution can be found in this situation if the path coordination [13] technique is used.

In various other tests operating our two robots in our narrow hallways, we frequently observed the emergence of solutions where robots sensibly coordinated their behavior, e.g., by waiting for each other. However, we also notice that with only two robots, these experiments do not evaluate the utility of our search algorithm in priority scheme space, since there exist only two such schemes. Unfortunately, we currently have only two physical robots available in our lab, so that the experiment could not be carried out with larger groups of robots.

6 Conclusions

This paper presented an approach to optimize priority schemes for arbitrary decoupled and prioritized path planning methods for groups of mobile robots. Our approach performs a randomized hill-climbing search in the space of priority schemes in order to find a solution and to minimize the overall path length. To guide the search, our approach exploits constraints extracted from the current task specification.

The approach has been implemented and tested on real robots. In addition, extensive simulations were performed to complement the physical robot experiments, The experiments suggest that our technique significantly decreases the number of failures in which no solution can be found, compared to a range of alternative approaches. Additionally, our approach leads to a significant reduction of the overall path length. A further advantage of our method lies in its general applicability. Although we applied our optimization technique only to two different baseline path-planning techniques in this paper, it is not limited to these two techniques. Rather, it can be used to find and optimize paths generated with arbitrary prioritized path-planning techniques.

Apart from the promising results presented in this paper, there are different aspects for future research. First, in the experiments carried out here, we assumed equal constant velocities for all robots. In practice, teams often are inhomogeneous and contain different types of robots with different average velocities which has to be taken into account. Furthermore, the techniques considered here provide no means to react to possible

deviations of the robots from their planned trajectories during the plan execution. For example, if one robot is delayed because unforeseen objects block its path, alternative plans for the robots might be more efficient. In such situations it would be important to have appropriate plan-revision techniques. Additionally, the delay of a single robot may result in a dead-lock during the plan execution. In this context, robot control systems require techniques to detect dead-locks while the robots are moving and to resolve such dead-locks appropriately.

References

1. K. Azarm and G. Schmidt. A decentralized approach for the conflict-free motion of multiple mobile robots. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1667–1674, 1996.
2. J. Barraquand, B. Langois, and J. C. Latombe. Numerical potential field techniques for robot path planning. *IEEE Transactions on Robotics and Automation, Man and Cybernetics*, 22(2):224–241, 1992.
3. J. Barraquand and J. C. Latombe. A monte-carlo algorithm for path planning with many degrees of freedom. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 1990.
4. Z. Bien and J. Lee. A minimum-time trajectory planning method for two robots. *IEEE Transactions on Robotics and Automation*, 8(3):414–418, 1992.
5. S. J. Buckley. Fast motion planning for multiple moving robots. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 1989.
6. C. Chang, M. J. Chung, and B. H. Lee. Collision avoidance of two robot manipulators by minimum delay time. *IEEE Transactions on Robotics and Automation, Man and Cybernetics*, 24(3):517–522, 1994.
7. M. Erdmann and T. Lozano-Perez. On multiple moving objects. *Algorithmica*, 2:477–521, 1987.
8. C. Ferrari, E. Pagello, J. Ota, and T. Arai. Multirobot motion coordination in space and time. *Robotics and Autonomous Systems*, 25:219–229, 1998.
9. L. Kavraki, P. Svestka, J. C. Latombe, and M. Overmars. Probabilistic road maps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, pages 566–580, 1996.
10. J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991. ISBN 0-7923-9206-X.
11. S. M. LaValle and S. A. Hutchinson. Optimal motion planning for multiple robots having independent goals. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 1996.
12. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. Technical report, Centre for Mathematics and Computer Science, 1989.
13. S. Leroy, J. P. Laumond, and T. Simeon. Multiple path coordination for mobile robots: A geometric algorithm. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.
14. P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proc. of the 5th International IPCO Conference*, pages 389–403, 1996.
15. H.P. Moravec and A.E. Elfes. High resolution maps from wide angle sonar. In *Proc. IEEE Int. Conf. Robotics and Automation*, pages 116–121, 1985.

16. N. J. Nilsson. *Principles of Artificial Intelligence*. Springer Publisher, Berlin, New York, 1982.
17. P. A. O'Donnell and T. Lozano-Perez. Deadlock-free and collision-free coordination of two robot manipulators. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 1989.
18. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard instances of satisfiability. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1992.
19. P. Sveska and M. Overmars. Coordinated motion planning for multiple car-like robots using probabilistic roadmaps. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 1995.
20. P. Tournassoud. A strategy for obstacle avoidence and its application to multi-robot systems. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, pages 1224–1229, 1986.
21. C. Warren. Multiple robot path coordination using artificial potential fields. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, pages 500–505, 1990.
22. S. Zilberstein and S. Russell. Approximate reasoning using anytime algorithms. In S. Natarajan, editor, *Imprecise and Approximate Computation*. Kluwer Academic Publishers, Dordrecht, 1995.

Possible Worlds Semantics for Credulous and Contraction Inference

Alexander Bochman

Computer Science Department
Holon Academic Institute of Technology, Israel
bochmana@hait.ac.il,
http://www.hait.ac.il/computers/staff/Bochman/index.htm

Abstract. A possible worlds semantics is suggested for a broad class of nonmonotonic inference relations, including not only traditional skeptical ones, but also credulous and contraction inference. The semantics could be used to provide a canonical framework for studying and comparing different kinds of nonmonotonic inference.

1 Introduction

One of the important approaches to nonmonotonic reasoning consists in describing associated nonmonotonic inference relations and their semantics. Thus, the so-called KLM theory [10,11] has suggested a semantic representation of preferential inference relations in terms of possible worlds models in which the world-states are ordered by a preference relation. In this framework, a nonmonotonic inference rule $A \mid\!\sim B$ was defined as saying that B should hold in all preferred world-states satisfying A.

The above mentioned preferential inference relations were designed to capture a skeptical approach to nonmonotonic reasoning, according to which, if there is a number of equally preferred alternatives, we infer only what is common to all of them. However, works in nonmonotonic reasoning have suggested also an alternative approach, usually called *credulous* or *brave* reasoning, in which each of the preferred alternatives is considered as an admissible solution to the nonmonotonic reasoning task. Many important reasoning problems in AI, such as diagnosis, abduction and explanation, are best seen as involving the search for particular preferred solutions. This idea is implicit also in the notion of an extension in default logic and its generalizations, as well as in similar constructs in modal nonmonotonic logics.

It turns out that preferential KLM models from [10] are not suitable for representing the above notion of credulous inference. In [2], an axiomatization and semantic representation of credulous inference has been suggested based on the notion of an epistemic state that can be viewed as a generalization of KLM models (see below). Epistemic states, however, are not possible worlds models, since their admissible states are labeled with deductively closed theories that are in general not complete.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 94–105, 2001.

In addition to brave nonmonotonic reasoning, a well-known correspondence between nonmonotonic inference and belief revision (see, e.g., [8]) suggests yet another, quite different kind of nonmonotonic inference relation that corresponds to a contraction operation in the theory of belief change. This kind of inference relations has been introduced in [3], and it has also received a semantic representation in the framework of epistemic states.

In this report we will show that the above 'non-standard' kinds of nonmonotonic inference can also be given a traditional, though slightly unusual, possible worlds semantics. In fact, the only change we must do to KLM models in order to obtain such semantics consists in considering the preference relation among worlds to be a *weak* partial order, in contrast to strict partial orders usually used for representing preferential inference. Using such weak orders, and especially the derived equivalence relation on worlds, will allow us to give an exact semantic description for the above kinds of inference in the framework of ordinary possible worlds models.

The possibility of such a possible worlds representation suggests an alternative, more standard, viewpoint on the above inference relations. In particular, it opens the possibility of using standard modal logic techniques for their study. Finally, it allows us to make quite transparent comparisons between different kinds of nonmonotonic inference.

1.1 Epistemic States

As a preparation, we describe in this section the notion of an epistemic state that has been used in [2,3] for representing all the above-mentioned kinds of nonmonotonic inference. Epistemic states stem from a quite common understanding according to which nonmonotonic reasoning uses not only known facts, but also *defaults*, or *expectations*, we have about the world[1]. These defaults are used as auxiliary assumptions that allow us to 'jump' to useful conclusions and beliefs that are not logical consequences of the facts alone. Such conclusions are defeasible and can be retracted when further facts become known to the reasoning agent. In addition, not all defaults are equally plausible or acceptable, and this creates, in turn, priorities and preferences among otherwise admissible combinations of defaults.

Since accepted combinations of defaults are primarily used for making inferences, admissible sets of defaults can be safely replaced by their deductive closures. So, instead of a prioritized collection of admissible sets of defaults, we can consider a set of deductively closed theories ordered by a preference relation. A slight generalization of this picture leads to the following notion of an epistemic state:

[1] We use here a 'naive' understanding of defaults as propositions (as in [8] and [14]), in contrast to formalizations of defaults made in default logic and some theories of nonmonotonic inference.

Definition 1. *An* epistemic state $\mathbb{E}$ *is a triple* $(S, \prec, l)$*, where* S *is a set of objects* admissible belief states*,* $\prec$ *is a* preference relation *on* S*, while* l *is a labeling function assigning each admissible state a deductively closed theory.*

If $s \prec t$ holds for two admissible belief states from $\mathbb{E}$, we will say that t is *preferred* to s. For the purposes of this report, the above preference relation can be safely taken to be a *strict partial order*.

Formally, epistemic states are similar to preferential models of Makinson [13] and cumulative models from [10][2]. However, we will use epistemic states in a somewhat different way. The main difference stems from the understanding that these states are *epistemic*; in other words, they normally do not involve objective facts and evidences, but only our beliefs, defaults and expectations. Accordingly, we are primarily interested not in (preferred) admissible states that satisfy a given evidence, but rather in admissible states that are *compatible* with the evidence. For example, Poole's abductive framework from [14] is representable as an epistemic state formed by all consistent combinations of defaults; the preference order amounts in this case simply to set inclusion. Then prediction and explanation in Poole's system is determined by maximal sets of defaults that are consistent with given facts.

An admissible state $s \in S$ will be said to *support* a proposition A if $A \in l(s)$, and *consistent with* A if $\neg A \notin l(s)$. The set of admissible states that do not support A will be denoted by $]A[$. Clearly, the set of admissible states that are consistent with A will coincide with $]\neg A[$ (while $]A[$ is, of course, the set of states consistent with $\neg A$).

Let P be an arbitrary set of admissible belief states from S. An admissible state $s \in P$ will be said to be *preferred in* P if there is no admissible state t in P such that $s \prec t$. A set P of admissible states will be called smooth (see [10]) if, for any $s \in P$, either s is preferred in P or there exists a preferred state t in P such that $s \prec t$. Finally, an epistemic state $\mathbb{E}$ will be called *smooth* if any set of belief states of the form $]A[$ is smooth.

As can be seen, the above definition of smooth epistemic states is different from that given in [10], since we require smoothness for sets of belief states that are consistent with some proposition. We will presume in what follows that our epistemic states are smooth. Note, however, that the smoothness requirement is trivial for finite epistemic states in which the preference relation is a partial order.

2 Skeptical Inference

Since epistemic states represent relatively stable default beliefs, in order to employ them in particular evidential situations, we should restrict our attention to admissible belief states that are consistent with the current facts, and choose preferred among them. The latter are used to support the conclusions and assump-

[2] Note that labeling with a deductively closed theory is equivalent to labeling with a set of worlds, as in [10].

tions we make about the situation at hand. Accordingly, all kinds of nonmonotonic inference are based on a two-step selection procedure: given a proposition E representing current evidence, we consider admissible belief states that are consistent with E and choose preferred elements in this set. Differences among various kinds of nonmonotonic inference will arise only at this point, due to different use we will make of these preferred belief states.

A *skeptical inference* with respect to an epistemic state is obtained if we decide that, given the set of preferred belief states consistent with the facts, we can reasonably infer only what is supported by each of these states. In other words, A will be a skeptical conclusion from the evidence E in a given epistemic state $\mathbb{E}$ if each preferred admissible belief set in $\mathbb{E}$ that is consistent with E, taken together with E itself, implies A. Or, in still other words,

Definition 2. *A is a* skeptical consequence *of E (notation $E \mid\!\sim A$) in an epistemic state if $E \rightarrow A$ is supported by all preferred admissible states in $]\neg E[$.*

A set of conditionals $A \mid\!\sim B$ that are valid in an epistemic state $\mathbb{E}$ in accordance with the above definition will be called a *skeptical inference relation determined by* $\mathbb{E}$. Accordingly, we will say that a set of conditionals forms a skeptical inference relation if it is determined by some epistemic state.

The above definition constitutes a straightforward generalization of the corresponding definition of prediction in Poole's abductive abductive framework (see [14]). It provides also a generalization of the notion of an expectation-based inference, given in [8].

Historical note. The above epistemic definition of conditionals is actually very old. In fact, the 'standard' definition of nonmonotonic inference, given in [10], derives from the relatively modern possible worlds theory of conditionals developed by Stalnaker and Lewis. The above definition, however, can be traced back to the era before the discovery of possible worlds, namely to Frank Ramsey and John S. Mill. In fact, this semantic definition can be seen as a particular variant of the *Ramsey test for conditionals* (see [7]). Though less familiar, it has also been used in the so-called 'premise-based' semantics for counterfactuals proposed by Veltman and Kratzer [17,9]. The relation between the two approaches to analyzing conditionals has been studied already by David Lewis in [12].

It turns out (see [2]) that the above semantic representation determines exactly preferential inference relations from [10]. Notice that preferential possible worlds models from [10] constitute a special case of epistemic states; namely, they correspond to epistemic states in which admissible states are labelled with 'worlds' (maximal consistent sets). In this respect, the completeness result proved in [10] says, in effect, that already such world-based epistemic states are sufficient for representing any preferential inference relation.

We will establish below a general correspondence between epistemic states and possible worlds models that will explain, to some extent, the above results. But first we describe the two alternative notions of nonmonotonic inference, mentioned in the introduction.

3 Credulous Inference

A credulous inference with respect to an epistemic state is defined by assuming that we can reasonably infer (or explain) conclusions that are supported by at least one preferred admissible state consistent with the facts. In other words, A will be a credulous conclusion from the evidence E in a given epistemic state $\mathbb{E}$, if at least one preferred admissible belief set in $\mathbb{E}$ that is consistent with E, taken together with E itself, implies A. In still other words,

Definition 3. *A is a* credulous consequence *of E (notation $E \mid\!\approx A$) in an epistemic state if $E \to A$ is supported by at least one preferred admissible state in $]\neg E[$.*

As before, a set of conditionals $A \mid\!\approx B$ that are valid in an epistemic state $\mathbb{E}$ will be called a *credulous inference relation determined by* $\mathbb{E}$.

There have been a few attempts in the literature to investigate the properties of credulous inference, mainly with negative conclusions that such an inference does not satisfy practically all 'respectable' rules (see, e.g., [4,5]). Thus, a distinctive feature of credulous reasoning is that it does not allow to conjoin different conclusions derivable from the same premises (because they might be grounded on different preferred solutions). In other words, credulous inference renders invalid the rule And.

(And) If $A \mid\!\sim B$ and $A \mid\!\sim C$, then $A \mid\!\sim B \wedge C$.

An important role of the rule And in the classification of inference systems has been pointed out already by Gabbay in [6].

In [2], an axiomatic characterization of credulous inference has been presented based on the approach to conditionals developed by van Benthem in [1]. The axiomatization amounted roughly to the removal of the above postulate And from the characterization of rational inference relations. The resulting axiomatization has been shown to be complete with respect to the above semantic notion of credulous validity:

Theorem 1. *An inference relation is credulous if and only if it coincides with the set of conditionals that are credulously valid in some epistemic state.*

The above result will be used below also for establishing completeness of credulous inference with respect to certain possible worlds semantics.

3.1 Permissive Inference

We will briefly describe now yet another kind of non-skeptical inference, namely permissive inference relations. It is intimately connected with the so-called *X-logics* suggested by Siegel and Forget in [16].

An inference relation is called *permissive* if it satisfies all the postulates of preferential (skeptical) inference except Cautious Monotony, which is weakened to the following rule:

(Conjunctive Cautious Monotony) If $A \mid\!\sim B \wedge C$, then $A \wedge B \mid\!\sim C$.

It turns out (see [2]) that permissive inference is an exact dual of a credulous inference under the following duality transformation that is familiar from the literature on conditional logics as a duality between ordinary conditionals and *might*-conditionals:

(Dual) $$A \mid\!\sim^{\diamond} B \quad \equiv \quad A \not\mid\!\sim \neg B \text{ or } A \mid\!\sim \mathbf{f}$$

Thus, if we apply this transformation to a credulous inference relation, we will obtain a permissive inference relation, and vice versa.

The above duality can be used for obtaining the properties of permissive inference by 'dualizing' corresponding properties of credulous inference. Thus, the semantic interpretation of credulous inference immediately gives us the following semantic characterization of permissive inference in epistemic states:

Definition 4. *A conditional $A \mid\!\sim B$ will be said to be* permissively valid *in an epistemic state $\mathbb{E}$ if any preferred admissible state consistent with A is consistent also with $A \wedge B$.*

A most plausible reading of a permissive conditional appears to be "*A is normally consistent with B*", since it asserts that all normal situations consistent with A are such that B could be added to them without losing consistency. Since the above semantic description corresponds precisely to the interpretation of the relation that is dual to credulous entailment, we immediately obtain

Theorem 2. *An inference relation is permissive if and only if it coincides with the set of conditionals that are permissively valid in some epistemic state.*

4 Contraction Inference

In the general correspondence between nonmonotonic inference and belief change, contraction inference corresponds to the basic operation of belief contraction. This augments the idea that belief change and nonmonotonic inference are "two sides of the same coin" and extends it to belief contractions.

Our earlier definition of skeptical inference with respect to epistemic states displays the latter as a composite notion. Namely, it says that $A \mid\!\sim B$ holds if and only if the implication $A \rightarrow B$ is supported by all preferred belief states that do not contain $\neg A$. This construction resembles quite closely the two-step construction of revision in the belief change theory. According to the latter, in order to revise a belief set with a new, possibly incompatible belief A, we should first contract $\neg A$ from the belief set, and then expand the result by adding A. This resemblance suggests that skeptical inference can be expressed using a more fundamental, or primitive, concept corresponding to the contraction operation in belief revision. This concept can be described as follows:

Definition 5. *B is a* contraction consequence *of A (notation $A \sim\!\!\!| B$) in an epistemic state if B is supported by all preferred admissible states in $]A[$.*

Thus, $A \sim\!\!\!| B$ holds if B is a plausible belief in the absence of A. We will call an expression of the form $A \sim\!\!\!| B$ a *contraction conditional* or, in short, a *contractional*. An informal reading of such contractionals will be *"In the absence of A, normally B"*.

Remark 1. Rules of the form "In the absence of A, accept B" are actually a simplest kind of *default rules* that constitute the subject matter of Reiter's default logic [15]. The relationship between contractionals and Reiter's default rules remains yet to be explored; presumably, it would be an important step on still a long way towards a future general theory of nonmonotonic reasoning. The similarity suggests, however, that a contractional $A \sim\!\!\!| B$ could be read as "*Unless A, B*", with the only reservation that we do not accept the usual presupposition associated with the latter expression, namely that A is by itself an unexpected (abnormal) condition.

Given the above definition of contraction inference, we can re-define now skeptical inference as follows:

$$A \mid\!\sim B \;\equiv\; \neg A \sim\!\!\!| A \rightarrow B$$

As can be seen, the above definition, coupled with the semantic definition of contraction inference, gives us exactly the definition of skeptical inference with respect to epistemic states. Accordingly, many properties of the latter can be analyzed already on the level of contraction inference relations.

Similarly to conditionals, a set of contractionals that are valid in an epistemic state $\mathbb{E}$ in accordance with the above definition will be called a *contraction inference relation* determined by $\mathbb{E}$. An axiomatic characterization of contraction inference relations and its completeness with respect to epistemic states have been given in [3]. The latter paper contains also a number of representation results for extensions of the contraction relation depending on various constraints imposed on epistemic states. These results cover a broad range of possible belief contraction functions including traditional AGM contractions, as well as contractions that do not satisfy the recovery postulate.

For completeness sake, we present below an axiomatic characterization of contraction inference. To begin with, general contraction relations are inference relations satisfying the following postulates:

Tautology	$A \sim\!\!\!| \mathbf{t}$
And	If $A \sim\!\!\!| B$ and $A \sim\!\!\!| C$, then $A \sim\!\!\!| B \wedge C$.
Right Weakening	If $B \vDash C$ and $A \sim\!\!\!| B$, then $A \sim\!\!\!| C$.
Extensionality	If $\vDash A \leftrightarrow C$ and $A \sim\!\!\!| B$, then $C \sim\!\!\!| B$.

Partial Antitony If $A \wedge B \sim\!\!\!\!\mid A$, then $A \wedge B \wedge C \sim\!\!\!\!\mid A$.

Cautious Antitony If $A \wedge B \sim\!\!\!\!\mid A$ and $B \sim\!\!\!\!\mid C$, then $A \wedge B \sim\!\!\!\!\mid C$.

Distributivity If $A \sim\!\!\!\!\mid C$ and $B \sim\!\!\!\!\mid C$, then $A \wedge B \sim\!\!\!\!\mid C$.

Cautious Monotony If $A \wedge B \sim\!\!\!\!\mid A \wedge C$, then $B \sim\!\!\!\!\mid C$.

The above postulates are still insufficient for giving a complete description of contraction inference with respect to epistemic states. We need also an additional postulate that fixes the result of 'impossible' contractions, such as contractions of tautologies. A simplest possible way of doing this is a 'classical' one, according to which if we are forced to disbelieve logical tautologies, we are allowed to believe anything[3]. This stipulation is actually presupposed by the above semantic definition of contraction inference with respect to epistemic states: when we are saying that $A \sim\!\!\!\!\mid B$ is valid in an epistemic state if B holds in all preferred belief states that do not satisfy A, this implies, in particular, that if A holds in all admissible states, then $A \sim\!\!\!\!\mid B$ will be (trivially) valid, for any B.

A general contraction relation will be called *simple* if it satisfies

Simple Failure If $A \sim\!\!\!\!\mid A$, then $A \sim\!\!\!\!\mid \mathbf{f}$.

Then a slight modification of the completeness result proved in [3] will give us the following

Theorem 3. *A contraction inference relation is simple iff it coincides with a set of contractionals valid in some epistemic state.*

5 Reflexive Possible Worlds Models

In this section we are going to show that all the above described kinds of non-monotonic inference can also be given a possible worlds semantics. More exactly, we will give a representation of these inference relations in terms of possible worlds models in which the accessibility (preference) relation is a pre- order.

Definition 6. *A* reflexive possible worlds model *is a triple* $\mathbb{W} = (W, l, \preccurlyeq)$*, where* W *is a set of states,* l *is a labeling function assigning each* $i \in W$ *a maximal deductively closed theory (a 'world'), and* $\preccurlyeq$ *is a pre-order on* W*.*

Given a pre-order $\preccurlyeq$ (i.e., a reflexive and transitive relation), we can immediately define the corresponding strict partial order in a well-known way:

$$s \prec t \;\equiv\; s \preccurlyeq t \text{ and } t \not\preccurlyeq s$$

Thus, reflexive models include, in a sense, preferential KLM models. This allows us to extend the terminology adopted for the latter to reflexive models. In

[3] In [3], a different, 'conservative' stipulation has been used, since it corresponded more closely to the behavior of contraction operations in common theories of belief change.

particular, we will continue to use the notions of a preferred state and smoothness as applied to the above (defined) relation $\prec$.

Weak partial order determines, however, a richer structure on possible worlds than a strict one. In particular, it allows us to express the idea that two states can be *equally preferred*:

$$s \sim t \ \equiv \ s \preccurlyeq t \text{ and } t \preccurlyeq s$$

This derived equivalence on states will play an essential role in representing non-standard inference relations, given below.

5.1 General Transformation

A general correspondence between epistemic states and reflexive possible worlds models will be established using a certain uniform transformation of epistemic states into their corresponding reflexive models.

Given an epistemic state $\mathbb{E} = (S, l, \prec)$, we define the corresponding reflexive possible worlds model $\mathbb{W}_{\mathbb{E}} = (W, l_0, \preccurlyeq)$ as follows:

- W is a set of all world-state pairs (α, s), where $s \in S$, and α is a world (maximal consistent set) such that $l(s) \subseteq \alpha$.
- The preference relation $\preccurlyeq$ on W is defined as follows:

$$(\alpha, s) \preccurlyeq (\beta, t) \text{ iff } s \prec t \text{ or } s = t.$$

- The labeling function l_0 on W is defined in an obvious way:

$$l_0((\alpha, s)) = \alpha.$$

As can be easily verified, $\preccurlyeq$ is a pre-order. Note that $(\alpha, s) \sim (\beta, t)$ holds if and only if $s = t$. In other words, states of $\mathbb{W}_{\mathbb{E}}$ are equally preferred in this order if and only if they correspond to the same admissible state of $\mathbb{E}$.

It turns out that the above defined reflexive model inherits many of the properties of the original epistemic state. It will be shown, in particular, that suitable definitions of credulous and contraction inference with respect to reflexive models will give us inference relations that will coincide with the corresponding inference relations determined by the source epistemic states. In this way, we will immediately obtain the completeness of reflexive possible worlds models with respect to these kinds of inference relations.

5.2 Possible Worlds Semantics for Credulous Inference

The following definition provides a characterization of credulous inference with respect to reflexive possible worlds models.

Definition 7. *A* credulously entails *B in a reflexive possible worlds model* $\mathbb{W}$ *if either A does not hold in any state of* $\mathbb{W}$*, or there exists a preferred state s supporting A such that all equally preferred states supporting A support also B. The set of conditionals that are credulously valid in* $\mathbb{W}$ *will be denoted by* $\approx_{\mathbb{W}}$.

The above definition says, in effect, that A credulously entails B in a reflexive possible worlds model if either A is false in all states of the model, or there exists a preferred state s satisfying A such that the classical implication $A \rightarrow B$ holds in all admissible states that are equally preferred relative to s.

To begin with, a straightforward check verifies that the above credulous inference satisfies all the axioms of a credulous inference relation.

Lemma 1. *For any reflexive possible worlds model* $\mathbb{W}$, $\approx_{\mathbb{W}}$ *is a credulous inference relation.*

Now we are going to show that the above possible worlds models are adequate for representing credulous inference. To this end we can use the following result:

Theorem 4. *If* $\mathbb{E}$ *is an epistemic state, then the corresponding reflexive possible worlds model* $\mathbb{W}_{\mathbb{E}}$ *determines the same credulous inference relation as* $\mathbb{E}$.

Since any credulous inference relation is determined by some epistemic state, the above theorem immediately gives a completeness of credulous inference with respect to reflexive possible worlds models:

Representation Theorem 1 *An inference relation is credulous if and only if it is generated by some reflexive possible worlds model.*

Semantics for permissive inference. Using the duality of credulous and permissive inference, we obtain the following characterization of permissive inference in terms of reflexive possible worlds models:

Definition 8. *A* permissively entails *B in a reflexive possible worlds model* $\mathbb{W}$ *if, for any preferred state supporting A there exists an equally preferred state that supports both A and B.*

Notice that if the classes of equally preferred states are singletons, then the above definition will coincide with the definition of preferential entailment in KLM models. In the general case, however, this definition gives a semantic characterization of permissive inference. Thus, the following result is actually an immediate consequence of the above representation theorem for credulous inference:

Representation Theorem 2 *An inference relation is permissive if and only if it is generated by some reflexive possible worlds model.*

5.3 Possible Worlds Semantics for Contraction Inference

Finally, we will give a representation of contraction inference in terms of reflexive models.

It turns out that contraction inference with respect to reflexive possible worlds models is definable as follows:

Definition 9. *B is a* contraction consequence of *A in $\mathbb{W}$ if either A holds in all states of $\mathbb{W}$, or there exists a preferred state s supporting $\neg A$ such that all equally preferred states support B.*

It can be easily verified that the above definition determines a simple contraction inference relation. Moreover, reflexive possible worlds models are also adequate for contraction inference. As before, this can be shown by proving that the reflexive possible worlds model $\mathbb{W}_{\mathbb{E}}$ corresponding to a given epistemic state $\mathbb{E}$ generates precisely the same contraction inference relation.

Theorem 5. *If $\mathbb{E}$ is an epistemic state, then its corresponding reflexive possible worlds model $\mathbb{W}_{\mathbb{E}}$ determines the same simple contraction inference relation as $\mathbb{E}$.*

Since any contraction inference relation is determined by some epistemic state, also in this case the above theorem gives a completeness of contraction inference with respect to reflexive possible worlds models.

Representation Theorem 3 *$\sim\!|$ is a contraction inference relation iff it is generated by some reflexive possible worlds model.*

6 Conclusions

The results described in this report show that many alternative notions of nonmonotonic inference can also be given a possible worlds semantics. Though we are not inclined to assign too much importance to such interpretations (as compared with epistemic states) the possibility of such a possible worlds semantics opens some new opportunities and perspectives in studying these kinds of nonmonotonic inference. Thus, possible worlds semantics are viewed by many as a canonical way of supplying semantics to logical notions. In this respect, the above results hopefully make the above non-standard notions of nonmonotonic inference more friendly and intelligible for a broad logical community. But most important, they suggest that well-developed methods of modal logic and possible worlds semantics could be used for studying these notions. In particular, there is a relatively straightforward way of constructing a (bi-)modal logic that would provide a syntactic description for reflexive possible worlds models. Then the above kinds of nonmonotonic inference can be described by certain formulas of such a modal logic. This is one of the topics for a further study.

References

1. J. Van Benthem. Foundations of conditional logic. *J. of Philosophical Logic*, 13:303–349, 1984.
2. A. Bochman. Credulous nonmonotonic inference. In T. Dean, editor, *Proceedings IJCAI'99*, pages 30–35, Stockholm, 1999. Morgan Kaufmann.
3. A. Bochman. Belief conraction as nonmonotonic inference. *Journal of Symbolic Logic*, 65:605–626, 2000.
4. S. Brass. On the semantics of supernormal defaults. In *Proceedings IJCAI-93*, pages 578–583, 1993.
5. C. Cayrol and M.-C. Lagasquie-Shiex. Non-monotonic syntax-based entailment: A classification of consequence relations. In C. Froidevaux and J. Kohlas, editors, *Symbolic and Qualitative Approaches to Reasoning and Uncertainty, ECSQARU'95*, pages 107–114, Fribourg, Switzerland, July 1995. Springer Verlag. Lecture Notes in AI, 946.
6. D. M. Gabbay. Theoretical foundations for non-monotonic reasoning in expert systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, Berlin, 1985. Springer-Verlag.
7. P. Gärdenfors. *Knowledge in Flux: Modeling the Dynamics of Epistemic States*. Bradford Books, MIT Press, 1988.
8. P. Gärdenfors and D. Makinson. Nonmonotonic inference based on expectations. *Artificial Intelligence*, 65:197–245, 1994.
9. A. Kratzer. Partition and revision: The semantics of counterfactuals. *J. of Philosophical Logic*, 10:201–216, 1981.
10. S. Kraus, D. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44:167–207, 1990.
11. D. Lehmann and M. Magidor. What does a conditional knowledge base entail? *Artificial Intelligence*, 55:1–60, 1992.
12. D. Lewis. Ordering semantics and premise semantics for counterfactuals. *Journal of Philosophical Logic*, 10:217–234, 1981.
13. D. Makinson. General patterns in nonmonotonic reasoning. In D. M. Gabbay and Others, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 3, Nonmonotonic and Uncertain Reasoning*, volume 2, pages 35–110. Oxford University Press, Oxford, 1994.
14. D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27–47, 1988.
15. R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.
16. P. Siegel and L. Forget. A representation theorem for preferential logics. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning. Proc. Fifth Int. Conference, KR'96*, pages 453–460. Morgan Kaufmann, 1996.
17. F. Veltman. Prejudices, presuppositions and the theory of conditionals. In J. Groenendijk and M. Stokhof, editors, *Amsterdam Papers on Formal Grammar*, volume 1, pages 248–281. Centrale Interfaculteit, Universiteit van Amsterdam, 1976.

The Point Algebra for Branching Time Revisited

Mathias Broxvall*

Department of Computer and Information Science
Linköpings Universitet
S-581 83 Linköping, Sweden
matbr@ida.liu.se

Abstract. Temporal reasoning with nonlinear models of time have been used in many areas of artificial intelligence. In this paper we focus on the model of branching time which has been proven successful for problems such as planning. We investigate the computational complexity of the point algebra for branching time extended with disjunctions and show that there are exactly five maximal tractable sets of relations. We also give an improved algorithm for deciding satisfiability of the point algebra with a time complexity comparable to that of path consistency checking algorithms.

1 Introduction

Many artificial intelligence systems include a component of temporal reasoning and many formalisms exist for describing and solving such problems. Two well known temporal languages are Allen's interval algebra [1] and the point algebra for totally ordered time [15]. In these algebras the basic entities are time intervals and time points, respectively. The satisfiability problem can be decided in polynomial time for the point algebra while it is NP-complete for Allen's interval algebra. Due to the computational hardness many attempts have been made to find tractable fragments of Allens's interval algebra.

However, Allen's interval algebra and other temporal algebras with a linear model of time is not sufficient in many applications. Instead, other models of time such as *branching* time have been suggested and studied in some detail. The model of branching time has proven especially successful for such applications. McDermott [12] has demonstrated the inadequacy of using only linear models of time in planning applications. Several logics based on branching time such as CTL and CTL* have been investigated earlier, see eg. the tutorial by Emerson and Srinivasan [9]. Furthermore, the point algebra for branching time has previously been examined by Düntsch *et al.* [8] from an algebraic point of view.

The point algebra for branching time has also been investigated from a computational perspective and k-consistency (for arbitrary k) has been shown insufficient for determining satisfiability for branching time by Hirsch [11]. Hirsch

* This research has been supported by the ECSEL graduate student program.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 106–121, 2001.

has also presented an algorithm running in $O(n^5)$ time for deciding satisfiability for this algebra. This can be compared to many other relational algebras where path consistency usually decides consistency for the tractable sets of relations. For instance, the point algebra for totally ordered time [15] and all tractable sets of relations for the point algebra for partially ordered time [5] as well as all known tractable sets of relations for RCC-5 [5] and RCC-8 [14] have this property.

In this paper we examine the point algebra for branching time from a computational perspective. We extend it with disjunctions for increased expressibility and give a complete classification of tractability. We do this for several reasons. First, simple constraint languages extended with disjunctions have been shown to have interesting properties and several examples of this have been given by Cohen *et al.* [6]. Secondly, disjunctions can compactly describe complex relations. Consider for example the ORD-Horn algebra [13] which contains 868 different relations. Defining it with the aid of disjunctions is very easy: ORD-Horn contains exactly the Allen relations which can be expressed by disjunctions of the form $x_1 \leq y_1 \vee x_2 \not= y_2 \vee \ldots \vee x_n \not= y_n$. Thus, tractability of ORD-Horn follows trivially from tractability of the point algebra for linear time extended with disjunctions. This approach has also proven successful for the point algebra for partially ordered time; a total classification of tractability has been given by Broxvall and Jonsson [4].

The main result of this paper is a total classification of tractability in the point algebra for branching time extended with disjunctions. Our results show that there exist exactly five maximal tractable sets of relations. Using these sets of relations we are not only able to solve problems containing these relations efficiently but can also backtrack upon these sets of relations for problem instances containing arbitrary relations and disjunctions. It should also be possible to form tractable sets of relations for the *interval* algebra for branching time. Additionally we give an improved algorithm for the full point algebra running in $O(nM(n))$ time where $M(n)$ is the time complexity of multiplying two $n \times n$ matrices. Coppersmith and Winograd [7] have shown that there exists an algorithm for matrix multiplication running in $O(n^{2.376})$ time. By using their matrix multiplication algorithm, our algorithm gets a time complexity of $O(n^{3.376})$ which should be compared to that of path consistency checking algorithms, $O(n^3)$. It is thus a significant improvement over Hirsch's algorithm.

Many relational algebras have been extended with disjunctions and the time complexity have been investigated. It is commonly found that extending a tractable set of relations containing disequality with disjunctions of disequality is again a new tractable set of relations. For instance, the point algebra for partially ordered and totally ordered time [4], RCC-5 and RCC-8 [5] have this property. Hence, one would expect tractability of the full point algebra for branching time extended with disjunctions of disequality. By showing that the opposite holds, we have one more example of the peculiar computational properties of this algebra.

The paper is structured as follows: Section 2 contains basic definitions and some auxiliary results used in the tractability and maximality proofs. This is

followed in Section 3 by the tractability results we need. We continue in the following section by giving NP-hardness proofs to yield a base of NP-complete problems which in turn is used in Section 5 to show our maximality result. Finally, the last section contains a brief summary of our results.

2 Preliminaries

The basic computational problem for the point algebra for branching time is that of deciding satisfiability. We begin by recalling the general constraint satisfaction problem CSPSAT. The satisfiability problem CSPSAT($\mathcal{S}$) for sets $\mathcal{S}$ of relations over a domain $\mathcal{D}$ is defined as follows:

Instance: A set V of variables over a domain $\mathcal{D}$ and a finite set Θ of constraints $\langle R, x_1, \ldots, x_n \rangle$, where $R \in \mathcal{S}$ is a relation of arity n and all $x_i \in V$.
Question: Is there a total function $f : V \to \mathcal{D}$ such that for each constraint $\langle R, x_1, \ldots, x_n \rangle \in \Theta$ the following holds: $\langle f(x_1), \ldots, f(x_n) \rangle \in R$.

The size of a problem instance can either be regarded as the total number of variables and constraints or (as is common in eg. path consistency) simply the total number of variables. We choose the later approach and define the size of a problem instance as the total number of variables since this gives a stronger result than the alternative definition.

For the point algebra for branching time, we are interested in CSPSAT problem instances over a specific domain and allow only certain relations. We write $\mathrm{SAT}_{\mathrm{br}}(\Gamma)$ to denote CSPSAT problem instances over this domain and with relations only from Γ. We define the domain $\mathcal{D}_{\mathrm{br}}$ as the points in the forest containing all finite trees infinitely many times. We choose this definition rather than restricting all points to have a common ancestor since it simplifies the proofs and otherwise change nothing. The basic relations over points in this domain are denoted by $<, >, =$ and $||$. Given arbitrary points x, y in $\mathcal{D}_{\mathrm{br}}$ we say that:

1. $x < y$ iff x precedes y in $\mathcal{D}_{\mathrm{br}}$.
2. $x > y$ iff y precedes x in $\mathcal{D}_{\mathrm{br}}$.
3. $x = y$ iff x, y are the same point.
4. $x \| y$ iff x, y belong to different branches or trees.

We will also refer to the point algebra for totally ordered time in some proofs. We write $\mathrm{SAT}_{\mathrm{to}}(\Gamma)$ to denote the CSPSAT problem instances of this domain where only relations from Γ are allowed. The set of basic relations for $\mathrm{SAT}_{\mathrm{to}}$ is the relations $<, >, =$.

Example 1. In the subset of $\mathcal{D}_{\mathrm{br}}$ (depicted in Figure 1) some of the relations holding between points are: $a < c$, $b \| c$, $b \| e$, $a < d$ and $c \| f$.

We take unions of the basic relations to form new binary relations. Sometimes we will use a short hand notation for such relations. We write eg. $\leq$, $\|$ and $\top$ instead of $< \cup =$, $\| \cup =$ and $< \cup > \cup \| \cup =$, respectively. We also use

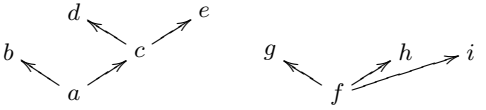


Fig. 1. A small subset of $\mathcal{D}_{\text{br}}$

the operators composition ($\gamma_1 \circ \gamma_2$), intersection ($\gamma_1 \cap \gamma_2$) and converse (γ^{-1}) of binary relations in the usual way. The composition table for the point algebra for branching time can be found in Table 1.

Given a problem instance Π of variables V and constraints C we say that a total function $f : V \to \mathcal{D}_{\text{br}}$ is an *interpretation* of Π. Furthermore, if f satisfies the constraints C then f is said to be a *model* of Π. To simply the proofs we need one further concept. A point $n \in \mathcal{D}_{\text{br}}$ related to some point in the image of f by $<$ or $>$ but not itself in the image is said to be a *redundant* point. If there exists no redundant points for f, ie. if f satisfies the following:

$$f(V) = \{a \in \mathcal{D}_{\text{br}} | \exists x \in V : a \leq f(x) \vee f(x) \leq a\}$$

we say that f is a *non-redundant* model of Π. Note that if Π has a model, then Π also has a non-redundant model since it is always possible to remove an arbitrary point from a tree and preserve the relations between remaining points.

Example 2. Let Π be a problem instance over five variables x_0, x_1, x_2, x_3, x_4 and the constraints: $x_0 < x_3, x_2 \not= x_3, x_4 \| x_3, x_2 \leq x_4, x_4 \leq x_2, x_1 \| x_4, x_0 <> x_1$ and $x_0 <> x_4$. Furthermore let, f_1, f_2 and f_3 be the following interpretations:

$$f_1(x_0) = a, f_2(x_0) = f, f_3(x_0) = a$$

$$f_1(x_1) = b, f_2(x_1) = g, f_3(x_1) = g$$
$$f_1(x_2) = d, f_2(x_2) = h, f_3(x_2) = d$$
$$f_1(x_3) = e, f_2(x_3) = i, f_3(x_3) = e$$
$$f_1(x_4) = d, f_2(x_4) = h, f_3(x_4) = d$$

where $a, \ldots, i$ are the point in $\mathcal{D}_{\text{br}}$ given in Figure 1. We have that f_1 and f_2 are models of Π but f_3 is not since the $x_0 <> x_1$ constraint is unsatisfied by f_3. Furthermore f_2 is a non-redundant model while f_1 is redundant since c is connected to the nodes of f_1 but not itself part of the image of f_1.

Next, we define the disjunction operator which enables us to form more general constraints from the basic relations.

Definition 1. *Let R_1, R_2 be relations of arity i, j and define the disjunction $R_1 \vee R_2$ of arity $i + j$ as follows:*

$$R_1 \vee R_2 = \{(x_1, \ldots, x_{i+j}) \in \mathcal{D}^{i+j} | (x_1, \ldots, x_i) \in R_1 \vee (x_{i+1}, \ldots, x_{i+j}) \in R_2\}$$

Table 1. The composition table for the point algebra for branching time

	$<$	$>$	$\|\|$	$=$
$<$	$<$	$<=>$	$< \|\|$	$<$
$>$	$\top$	$>$	$\|\|$	$>$
$\|\|$	$\|\|$	$> \|\|$	$\top$	$\|\|$
$=$	$<$	$>$	$\|\|$	$=$

To give a concrete example of how the disjunction operator is used consider the following:

Example 3. Let $D = \{0, 1\}$ and the relations $\mathsf{And} = \{\langle 1, 1\rangle\}$ and $\mathsf{Xor} = \{\langle 0, 1\rangle, \langle 1, 0\rangle\}$ be given. The disjunction of And and Xor is:

$$\mathsf{And} \vee \mathsf{Xor} = \left\{ \begin{array}{l} \langle 0,0,0,1\rangle, \langle 0,1,0,1\rangle, \langle 1,0,0,1\rangle, \langle 1,1,0,1\rangle, \\ \langle 0,0,1,0\rangle, \langle 0,1,1,0\rangle, \langle 1,0,1,0\rangle, \langle 1,1,1,0\rangle, \\ \langle 1,1,0,0\rangle, \langle 1,1,0,1\rangle, \langle 1,1,1,0\rangle, \langle 1,1,1,1\rangle \end{array} \right\}$$

We see that the constraint x And $y \vee x$ Xor z encoded by $\langle \mathsf{And} \vee \mathsf{Xor}, x, y, x, z\rangle$ is satisfied when x, y and z have been instantiated to, for instance $1, 0, 0$ respectively.

We continue by defining the disjunction over sets of relations. The disjunction of two sets of relations is the set containing the disjunction of every pair of relations in the original set as well as the original relations themselves. It is sensible to include the original relations since one wants to have the choice of using the disjunctions or not.

Definition 2. *Let Γ_1, Γ_2 and Δ be sets of relations. We define disjunctions of two sets of relations $\Gamma_1 {}^{\times}_{\vee} \Gamma_2$ and disjunctions over a single set of relations Δ^i, Δ^* as follows:*

$$\Gamma_1 {}^{\times}_{\vee} \Gamma_2 = \Gamma_1 \cup \Gamma_2 \cup \{R_1 \vee R_2 \mid R_1 \in \Gamma_1, R_2 \in \Gamma_2\}$$

$$\Delta^1 = \Delta \quad \Delta^{i+1} = \Delta^i {}^{\times}_{\vee} \Delta$$

$$\Delta^* = \bigcup_{i=1}^{\infty} \Delta^i$$

Finally, we introduce the 1-independence property as defined by Cohen *et al.* [6]. For simplicity we write independence rather than 1-independence. This concept will be used extensively for showing tractability results.

Definition 3. *For any sets of relations Γ and Δ, we say that Δ is independent with respect to Γ if the following holds: For any set of constraints C in* CSPSAT$(\Gamma \cup \Delta)$*, C has a solution whenever every $C' \subseteq C$, which contains at most one constraint whose constraint relation belongs to Δ has a solution.*

Table 2. Tractable classes

	Γ_A	Γ_B	Δ_B	Δ_C	Γ_D	Δ_D	Γ_E	Δ_E
$<$	•	•					•	
$\leq$	•	•		•			•	
$<>$	•	•	•				•	
$<=>$	•	•	•	•			•	
$\|$	•				•	•	•	
$\underline{\|}$	•			•	•	•		
$=$	•	•		•	•		•	
$/=$	•	•	•		•	•	•	•
$< \|$	•				•	•	•	
$\leq \|$	•			•	•	•		

Example 4. We see that $\{=\}$ is not independent of $\{=, /\not=\}$ since the constraints $C : x = y, y = z, x \not= z$ is not satisfiable although the constraints $C' : x = y, x \not= z$ and $C'' : y = z, x \not= z$ are both satisfiable. Note that $\{=\}$ is independent of itself since all problem instances containing only equality constraints are satisfiable.

The following theorem proven by Cohen *et al.* [6] provides a useful mechanism for determining satisfiability of disjunctions of relations.

Theorem 1. $\mathrm{CSPSAT}(\Gamma^{\veebar}\Delta^*)$ *is tractable for any sets of relations* Γ *and* Δ*, such that* $\mathrm{CSPSAT}(\Gamma \cup \Delta)$ *is tractable and* Δ *is independent with respect to* Γ*.*

3 Tractability Results

In this section we will show that the sets of relations defined below (where $\Gamma_A, \ldots, \Delta_E$ are defined in Table 2) are tractable. The same sets of relations will also be proven to be the unique maximal sets of tractable relations for $\mathrm{SAT}_{\mathrm{br}}$ in Section 5.

$$\mathcal{T}_A = \Gamma_A \qquad \mathcal{T}_B = \Gamma_B{}^{\veebar}\Delta_B^* \qquad \mathcal{T}_C = \Delta_C^*$$

$$\mathcal{T}_D = \Gamma_D{}^{\veebar}\Delta_D^* \qquad \mathcal{T}_E = \Gamma_E{}^{\veebar}\Delta_E^*$$

We begin by showing tractability of $\mathcal{T}_A$ and by proposing an algorithm running in $O(nM(n))$ time, where $M(n)$ denotes the time complexity of multiplying two $n \times n$ matrices. The algorithm can be found in Figure 2. Using the $O(n^{2.376})$ algorithm for matrix multiplication proposed by Coppersmith and Winograd [7] we see that our algorithm is a significant improvement over the $O(n^5)$ algorithm given by Hirsch [11].

We use the notion of components in the algorithm in the following sense: x, y are in the same component iff there exists a path between x and y. Note that the concept of components are only defined for undirected graphs. In order to simplify the algorithm we assume that the set of constraints for problem

```
1  Algorithm BRANCH(Π)
2  Input: Problem instance Π = ⟨V, C⟩ of Γ_A
3  Let G be the undirected graph ⟨V, ∅⟩
4  for each constraint ⟨R, x, y⟩ ∈ C such that ∥ /⊆R do
5    add the edge {x, y} to G
6  Partition G into components V_1, ..., V_k
7  Partition C into C_1, ..., C_k such that:
8  C_i = {⟨R, x, y⟩ ∈ C | x, y ∈ V_i}
9  for each component V_i do
10   Let G_i be the directed graph ⟨V_i, ∅⟩
11   for each c = ⟨R, x, y⟩ ∈ C_i such that </⊆R do
12     add the (directed) edge ⟨x, y⟩ to G_i
13   Let M be the adjacency matrix of the transitive
14   and reflexive closure of G_i
15   N ← new empty matrix indexed by V_i
16   for each ⟨R, x, y⟩ ∈ C_i such that =/⊆R do
17     N[x, y] ← 1
18   P ← M · N
19   root ← ⊥
20   for each x ∈ V_i do
21     if ∀y ∈ V_i : P[x, y] = 0 ∨ M[x, y] = 0 then
22       root ← x
23   if root /=⊥ then
24     V_i' ← V_i − {c | M[root, c] = 1}
25     C_i' ← {⟨R, x, y⟩ ∈ C_i | x, y ∈ V_i'}
26     if BRANCH(⟨V_i', C_i'⟩) rejects then reject
27   else reject
28 accept
```

Fig. 2. Algorithm for determining satisfiability for Γ_A

instances are closed under converse. We also use expressions such as $</\subseteq R$ for which relations should be considered as sets of tuples, eg. we have $<\subseteq\leq$.

The algorithm works by first partitioning the problem instance into sets of variables which can be mapped to disjoint trees, ie. all constraints between the partitions include the relation $\|$. This is done by partitioning the undirected graph G which contains an edge $\{x, y\}$ iff there exists a constraint $\langle R, x, y\rangle$ disallowing $\|$ into its components. Next, the algorithm tries to identify a variable *root* which can be mapped to the root node for each partition. Note that there exists exactly one such node for each satisfiable partition since each partition maps to a distinct tree.

Let x_1 be a candidate to be mapped to the root node in a satisfiable problem instance. If there exists a chain of constraints $x_1\ R_1\ x_2\ \cdots\ x_{n-1}\ R_n\ x_n$ such that $</\subseteq R_i$ we must also map $x_2, \ldots, x_n$ to the root node. Existence of the constraints $x_1\ R_1\ x_2\ \cdots\ x_{n-1}\ R_n\ x_n$ for each pair of variables x_1, x_n is checked by looking at the adjacency matrix M of the transitive closure for the graph

G_i constructed by the algorithm for every partition V_i. If $M[x, y] = 1$ and x is mapped to the root node then so must y.

Thus, for x to be mapped to the root node there must not exist a pair of variables which both are identified with x and for which we have a constraint disallowing the equality relation, ie. there must not exist nodes y, z and a constraint $c = \langle R, y, z\rangle$ such that $= \not\subseteq R$, $M[x, y] = 1$ and $M[x, z] = 1$. Conveniently, existence of z and c such that $= \not\subseteq R$ and $M[x, z] = 1$ can be computed by matrix multiplication. This is done in step 18, where $P[x, y] = 1$ iff there exists such a variable z and constraint c. After identifying a variable *root* which can be mapped to the root node, every variable which must be identified with *root* is removed from the partition and the algorithm is recursively called for this new problem instance.

Before proving correctness of the algorithm we recall the definition of matrix multiplication.

$$C = A \cdot B \Leftrightarrow C[i, j] = \Sigma_{k=1}^{n} A[i, k] \cdot B[k, j] \tag{1}$$

Lemma 1. *If Algorithm* Branch *rejects a problem instance then it is not satisfiable.*

Proof: Assume to the contrary that there exists a satisfiable problem instance Π which is rejected by the algorithm. Then there exists a component i of a subset of the original problem instance for which the algorithm rejects on line 27. Note that rejection on line 26 only occurs if the algorithm rejected on 27 after some recursions. Let Π' denote the problem instance $\langle V_i, C_i\rangle$ where V_i, C_i is the variables and constraints making the algorithm reject on line 27. Trivially, the algorithm rejects also Π'. Note that Π' is a subinstance of Π and, hence satisfiable.

Since Π' is satisfiable there exist a non-redundant model f of Π'. Assume that there exists more than one minimal point in the image of Π' under f and let $f(x)$ and $f(y)$ denote two such points. Note that minimal points in $\mathcal{D}_{\mathrm{br}}$ are the roots of distinct trees. Since the graph G constructed by the algorithm for Π' contains only one component then there exists a path of constraints $x\ R_1\ x_1\ \cdots\ x_n\ R_{n+1}\ y$ such that $\| \not\subseteq R_1, \ldots, R_{n+1}$. Since $f(x)$ and $f(y)$ are the root of different trees we obviously have some x_i, x_{i+1} whose images lies in different trees thus violating the constraint $x_i\ R_{i+1}\ x_{i+1}$. Contradiction, and we have only one minimal point which must be the root of all other points in the image in Π'.

Let x denote a variable in Π' such that $f(x)$ is the minimal point. Since the algorithm rejects we know that there exists some $y \in V_i$ such that $P[x, y] \geq 1$ and $M[x, y] = 1$. From the definition of matrix multiplication (1) follows the equations (2) and (3) which obviously are equivalent.

$$P[x, y] = \sum_{z \in V_i} M[x, z] \cdot N[z, y] \geq 1 \tag{2}$$

$$\exists z \in V_i : M[x, z] = N[z, y] = 1 \tag{3}$$

Hence, we have $z \in V_i$, $c = \langle R, z, y\rangle \in C_i$ such that $M[x,z] = 1$ and $= \not\subseteq R$. Since $M[x,y] = 1$, we have a chain of constraints $x\ R_1\ y_1\ R_2\ \cdots\ R_{n+1}\ y$ such that $< \not\subseteq R_i$. Note that $f(x) \leq f(y_i)$ and hence, $f(x) = f(y_1)$ and $f(y_1) \leq f(y_i)$ which gives $f(x) = f(y_1) = f(y_2)$ and $f(y_2) \leq f(y_i)$ etc. This leads to $f(x) = f(y_i) = f(y)$. Analogously, $f(x) = f(z_i) = f(z)$. Contradiction, since $f(z) = f(x) = f(y)$ violates the constraint c. Thus, Π' and Π are not satisfiable. □

Having proven that the algorithm rejects only unsatisfiable instances we will now demonstrate that the algorithm correctly identifies the satisfiable problem instances.

Lemma 2. *Algorithm* Branch *only accepts satisfiable problem instances.*

Proof: We prove the result by induction over the number of variables in the given problem instance. Obviously, all problem instances accepted by the algorithm containing zero variables are satisfiable. Assume that acceptance of the algorithm implies satisfiability for all problem instances of size n or less. Let Π be a problem instance of size $n+1$ which is accepted by the algorithm. We construct an interpretation f of the problem instance as follows:

1. For each component i, let f'_i denote a model of $\langle V'_i, C_i\rangle$. Note that $|V'_i| < |V|$ since at least one variable is removed on line 24. Existence of f'_i follows from the induction hypothesis.
2. For each component i, introduce a fresh root node t to each f'_i. We define an interpretation f_i of $\langle V_i, C_i\rangle$ as follows:

$$f_i(x) = \begin{cases} t & \text{iff } M[root, x] = 1 \\ f'_i(x) & \text{otherwise} \end{cases}$$

3. Define f as the union of the (disjoint) interpretations f_i.

We continue by demonstrating that each constraint $C = \langle R, x, y\rangle$ is satisfied by f. Assume $x \in V_i$ and $y \in V_j$. We have two cases.
Case 1: If $i = j$ there are four further cases to analyze. If $M[root, x] = M[root, y] = 0$ then C is included in $\langle V_i, C_i\rangle$ and hence, satisfied by f'_i,f_i and thus f. When $M[root, x] = M[root, y] = 1$ we have $P[root, y] = 0$ which implies $N[x, y] = 0$. Hence, we have $= \subseteq R$ and thus, C is satisfied by f_i and f. Finally, when $M[root, x] = 1$ and $M[root, y] = 0$ we have $< \subseteq R$ since there otherwise would exist a path from *root* to y in the graph G_i constructed by the algorithm. Hence, f_i and f satisfies C. The case $M[root, x] = 0$ and $M[root, y] = 1$ can be shown analogously.
Case 2: $i \neq j$. Since the algorithm has partitioned x and y into different components we know that $\| \subseteq R$ and c is satisfied by f. □

By demonstrating that the algorithm runs in polynomial time we can now conclude that the satisfiability problem for $\mathcal{T}_A$ is tractable. This is done in the following theorem.

Theorem 2. $\mathrm{SAT}_{\mathrm{br}}(\mathcal{T}_A)$ *is tractable.*

Proof: We have previously shown that algorithm BRANCH correctly decided satisfiability for $\mathrm{SAT}_{\mathrm{br}}(\mathcal{T}_A)$ problem instances. It is now only a matter of demonstrating that algorithm BRANCH runs in polynomial time. We begin by noting that the initial graph partitioning can be performed in $O(n^2)$ steps by applying a standard graph partitioning algorithm, see eg. Baase [2]. Also, the transitive closure of G_i on line 13 can be computed in the same time as the boolean matrix multiplication on line 18 [2]. Hence, steps 10-26 can all be performed within $O(M(n_i))$ time. The following polynomial is an estimate of the total running time of algorithm BRANCH:

$$f(n) \leq c_1 n^2 + \Sigma_{i=1}^{k} (c_2 M(n_i) + f(n_i - 1))$$

for some constants c_1, c_2 and where n_i denotes the size of component V_i. Note that we have $n_1 + \ldots + n_k = n$. Furthermore, for all $c \geq 0$ we have that $n_1^c + \cdots + n_k^c \leq (n_1 + \cdots + n_k)^c$. Hence, $\Sigma_{i=1}^{k} c_2 M(n_i) \leq c_2 M(\Sigma_{i=1}^{k} n_i) = c_2 M(n)$. This gives us: $f(n) \leq (c_1 + c_2)M(n) + \Sigma_{i=1}^{k} f(n_i - 1)$ which can be shown to evaluate to a polynomial with positive coefficients by an induction over n. Hence,

$$f(n) \leq (c_1 + c_2)M(n) + f(n-1) \leq (c_1 + c_2) n M(n)$$

□

We will now proceed to the four other tractable sets of relations. We begin by noting that $\mathcal{T}_B$ and $\mathcal{T}_C$ is tractable.

Theorem 3. *The satisfiability problem for* $\mathcal{T}_B$ *and* $\mathcal{T}_C$ *is tractable.*

Proof: The same sets of relations have previously been investigated and proven tractable for the point algebra for partially ordered time [4]. We note that every problem instance of $\mathrm{SAT}_{\mathrm{br}}(\mathcal{T}_B)$ and $\mathrm{SAT}_{\mathrm{br}}(\mathcal{T}_C)$ is satisfiable iff it has a totally ordered model, ie. a model f such that for all x, y holds $f(x) <=> f(y)$. Hence, we can solve $\mathrm{SAT}_{\mathrm{br}}(\mathcal{T}_B)$ and $\mathrm{SAT}_{\mathrm{br}}(\mathcal{T}_C)$ problem instances as problem instances of the point algebra for partially ordered time, which can be done in polynomial time. □

In our next theorem we demonstrate the tractability of $\mathcal{T}_D$.

Theorem 4. *The satisfiability problem for* $\mathcal{T}_D$ *is tractable.*

Proof: The following two steps is all that is needed for an algorithm determining satisfiability for problem instances Π of $\mathrm{SAT}_{\mathrm{br}}(\Gamma_D)$.

1. For each constraint $c = \langle =, x, y \rangle$, remove c and replace all occurrences of y in Π with x.
2. If there exists a constraint $\langle R, x, x \rangle$ such that $= \not\subseteq R$ then reject, else accept.

If the algorithm above rejects an instance, it is clearly not satisfiable. Otherwise, it is satisfiable since all remaining variables can be mapped to root nodes in disjoint trees. By an analysis of the algorithm it is obvious that Δ_D is independent of Γ_D and hence, $\Gamma_D \dot{\vee} \Delta_D^*$ is tractable. □

Theorem 5. $\mathrm{SAT}_{\mathrm{br}}(\mathcal{T}_E)$ *is tractable.*

Proof: First, let $\Gamma'_E = \{\leq, \neq, \|\}$. Broxvall and Jonsson [3] have constructed an algorithm running in polynomial time for deciding satisfiability of $\mathrm{SAT}_{\mathrm{br}}(\Gamma'_E)$ which is similar to the Branch algorithm presented in this paper. By a straightforward analysis of this algorithm, it can be proven that $\neq$ is independent of Γ'_E and hence, $\Gamma'_E \overset{\times}{\vee} \{\neq\}$ is tractable.

Next, a polynomial reduction from Γ_E instances to Γ'_E instances has been given by Broxvall and Jonsson [3]. By analyzing these proofs, it is evident that we can reduce $\Gamma_E \overset{\times}{\vee} \{\neq\}$ to $\Gamma'_E \overset{\times}{\vee} \{\neq\}$ in polynomial time. □

4 Intractability Results

This section contains the NP-completeness results which are needed for the main result. Note that for all sets of relations considered in this paper, the satisfiability problem is in NP. For the NP-hardness proofs we need the definition of the NP-complete problem 3-Colourability [10]:

Instance: Undirected graph $G = \langle V, E \rangle$
Question: Is G 3-colourable, i.e., does there exist a function $f : V \to \{1, 2, 3\}$ such that $\forall \langle u, v \rangle \in E : f(u) \neq f(v)$

There are large similarities between the four NP-completeness proofs provided in this section. Although the first one can be thought of as a general template for the construction of the other NP-completeness proofs we provide the other proofs since they give a good insight into the branching time model.

Lemma 3. $\mathrm{SAT}_{\mathrm{br}}(\{\| \vee \|, <=>\})$ *is NP-complete.*

Proof: We exhibit a reduction from 3-Colourability. Arbitrarily choose an undirected graph $G = \langle V, E \rangle$ such that $V = \{v_1, \cdots, v_n\}$. We construct an instance Π of $\mathrm{SAT}_{\mathrm{br}}(\{\| \vee \|, <=>\})$ which is satisfiable iff G is colourable with three colours. For each vertex v_i add the following constraints:

$$a_i <=> b_i <=> c_i <=> a'_i \tag{4}$$

$$a'_i <=> b'_i <=> c'_i <=> a_i \tag{5}$$

$$a_i \| a'_i \vee b_i \| b'_i \ , \ a_i \| a'_i \vee c_i \| c'_i \ , \ b_i \| b'_i \vee c_i \| c'_i$$

It can easily be seen that a problem instance given by the constraints (4-5) and $n = 1$ is satisfiable and remains so when we add one or two, but not three, of the constraints $f(a) \| f(a'), f(b) \| f(b'), f(c) \| f(c')$. Since the problem instance Π constructed so far is satisfiable we have that for every model f of Π, exactly one of $f(a_i) \| f(a'_i), f(b_i) \| f(b'_i)$ or $f(c_i) \| f(c'_i)$ does not hold. Our intention with the construction is that for each variable v_i, it holds that $f(a_i) <=> f(a'_i)$ iff v_i can be coloured with the first colour and so on for the other "colours" b_i and

c_i. Note that a vertex may have a choice of several colours in which case any of the colours can be chosen. For each edge $\langle v_i, v_j \rangle$ in G add the constraints:

$$a_i \| a'_i \vee a_j \| a'_j \ , \ b_i \| b'_i \vee b_j \| b'_j \ , \ c_i \| c'_i \vee c_j \| c'_j$$

which ensures that v_i and v_j are not assigned the same colour. The resulting set of constraints can be computed in polynomial time, it is an instance of $\mathrm{SAT}_{\mathrm{br}}(\{\| \vee \|, <=>\})$ and it is satisfiable iff G is 3-colourable. □

Lemma 4. $\mathrm{SAT}_{\mathrm{br}}(\{\neq \vee \neq, \underline{\|}, <>\})$ *is NP-complete.*

Proof sketch: By adding the constraints $a_i \underline{\|} a'_i, b_i \underline{\|} b'_i, c_i \underline{\|} c'_i$ to those given by (4-5) in the proof of Lemma 3 we can substitute all occurrences of $\|$ by $/=$. □

Lemma 5. *Let* $R_1, R_2 \in \{<>, <=>\}$ *then* $\mathrm{SAT}_{\mathrm{br}}(\{R_1 \vee R_2, \|\})$ *is NP-complete.*

Proof sketch: The proof is very similar to the proof of Lemma 3. For each vertex the set of constraints should contain:

$$a \| b' \ , \ b \| c' \ , \ c \| a'$$

$$a \ R_1 \ b \ \vee \ b \ R_2 \ c \quad b \ R_1 \ c \ \vee \ c \ R_2 \ a \quad c \ R_1 \ a \ \vee \ a \ R_2 \ b$$

Which enforces that at least one of the a, a', b, b' or c, c' pairs will be related by the relation $\|$ which also determines the colour of the vertex. The constraints corresponding to the edges of G is of the same form as in lemma 3 but using the R_1 and R_2 constraints. □

Lemma 6. $\mathrm{SAT}_{\mathrm{br}}(\{\underline{\|} \vee \underline{\|}, <, <=>\})$ *is NP-complete.*

Proof: Again, we make a reduction from the NP-complete problem 3-Colourability. Arbitrarily choose an undirected graph $G = \langle V, E \rangle$ such that $V = \{v_1, \cdots, v_n\}$. We construct an instance Π of $\mathrm{SAT}_{\mathrm{br}}(\{\underline{\|} \vee \underline{\|}, <, <=>\})$ which is satisfiable iff G is colourable with three colours. For each vertex v_i, add the following constraints:

$$a_i <=> a'_i, b_i <=> b'_i, c_i <=> c'_i \ , \ a'_i < b_i, b'_i < c_i, c'_i < a_i$$

$$a_i \underline{\|} a'_i \vee b_i \underline{\|} b'_i \ , \ a_i \underline{\|} a'_i \vee c_i \underline{\|} c'_i \ , \ b_i \underline{\|} b'_i \vee c_i \underline{\|} c'_i$$

The problem instance constructed so far is satisfiable and for each model f exactly two of the following relations holds: $f(a_i) = f(a'_i), f(b_i) = f(b'_i)$ or $f(c_i) = f(c'_i)$. Note that the variable pair not related by equality will be related by $<>$ because of the three first constraints. Again, our intention is to assign colours to the vertices by the models constructed for Π. We say that a variable v_i can be coloured by the first colour when $f(a_i) \not= f(a'_i)$ and so forth for b_i and c_i. For each edge $\langle v_i, v_j \rangle$ in G add the constraints:

$$a_i \underline{\|} a'_i \vee a_j \underline{\|} a'_j \ , \ b_i \underline{\|} b'_i \vee b_j \underline{\|} b'_j \ , \ c_i \underline{\|} c'_i \vee c_j \underline{\|} c'_j$$

which ensures that v_i and v_j are not assigned the same colour. The resulting set of constraints can be computed in polynomial time, it is an instance of $\mathrm{SAT}_{\mathrm{br}}(\{\underline{\|} \vee \underline{\|}, <, <=>\})$ and it is satisfiable iff G is 3-colourable. □

Theorem 6. *The satisfiability problem is NP-complete for the following sets of relations:*

1. $\{<\}^{\times}_{\diamond}\{<\}$
2. $\{=\}^{\times}_{\diamond}\{=\} \cup \{R\}$ *where* $=/\in R$.
3. $\{R_1\}^{\times}_{\diamond}\{R_2\} \cup \{< \,||, <\}$ *where* R_1, R_2 *are* $<>$ *or* $<=>$.
4. $\{\| \vee \|, <\}$,$\{\underline{\|} \vee \underline{\|}, <, <=>\}$ *and* $\{\neq \vee \neq, \underline{\|}, <=>\}$

Proof: For the two first points we note that $\mathrm{SAT}_{\mathrm{to}}(\{<\}^{\times}_{\diamond}\{<\})$ and $\mathrm{SAT}_{\mathrm{to}}(\{=\}^{\times}_{\diamond}\{=\} \cup \{R\})$ have been proven NP-complete previously by Broxvall and Jonsson [4]. The reduction to the corresponding branching time satisfiability problem is trivial. The remaining sets of relations have been proven NP-complete in this section. □

5 Maximality

We will now demonstrate that the five sets of relations proven tractable in Section 3 are the only maximal tractable sets. In order to do so we need a concept of maximality for relational algebras with disjunctions and we use the definition in Broxvall *et al.* [5]. Let Γ be a set of disjunctive relations constructed from a set $\mathcal{B}$ of binary relations by applying the $^{\times}_{\diamond}$ operator. We say that Γ is a *maximal tractable set* iff Γ is tractable and for every set $X \not\subseteq \Gamma$ of relations which can be constructed by the relations in $\mathcal{B}$ and $^{\times}_{\diamond}$, $\Gamma \cup X$ is intractable. We need a number of equivalence results.

Lemma 7. *The following problems are equivalent up to polynomial-time reductions.*

1. $\mathrm{SAT}_{\mathrm{br}}(\Gamma)$ *and* $\mathrm{SAT}_{\mathrm{br}}(\Gamma')$ *where* Γ' *is the closure of* Γ *with respect to converse, composition and intersection.*
2. $\mathrm{SAT}_{\mathrm{br}}(\{< \vee R\} \cup \Gamma)$ *and* $\mathrm{SAT}_{\mathrm{br}}(\{< \vee R, R \vee R\} \cup \Gamma)$.
3. $\mathrm{SAT}_{\mathrm{br}}(\{= \vee R, R'\} \cup \Gamma)$ *and* $\mathrm{SAT}_{\mathrm{br}}(\{= \vee R, R', R \vee R\} \cup \Gamma)$ *where* $=/\subseteq R'$.
4. $\mathrm{SAT}_{\mathrm{br}}(\{\| \vee R, R'\} \cup \Gamma)$ *and* $\mathrm{SAT}_{\mathrm{br}}(\{\| \vee R, R', R \vee R\} \cup \Gamma)$ *where* R' *is* $<>$ *or* $<=>$.
5. $\mathrm{SAT}_{\mathrm{br}}(\{\underline{\|} \vee R, < \|, R'\} \cup \Gamma)$ *and* $\mathrm{SAT}_{\mathrm{br}}(\{\underline{\|} \vee R, < \|, R', R \vee R\})$ *where* R' *is* $<>$ *or* $<=>$.
6. $\mathrm{SAT}_{\mathrm{br}}(\{\neq \vee R, \|, R\} \cup \Gamma)$ *and* $\mathrm{SAT}_{\mathrm{br}}(\{\neq \vee \neq, /\Rightarrow \vee R, \|, R\} \cup \Gamma)$ *whenever* R *is the relation* $<>$ *or* $<=>$*;*
7. $\mathrm{SAT}_{\mathrm{br}}(\{\gamma_1 \vee \gamma_2, \gamma_1 \vee \gamma_3\} \cup \Gamma)$, $\mathrm{SAT}_{\mathrm{br}}(\{\gamma_1 \vee \gamma_2, \gamma_1 \vee \gamma_3, \gamma_1 \vee (\gamma_2 \circ \gamma_3)\} \cup \Gamma)$ *and* $\mathrm{SAT}_{\mathrm{br}}(\{\gamma_1 \vee \gamma_2, \gamma_1 \vee \gamma_3, \gamma_1 \vee (\gamma_2 \cap \gamma_3)\} \cup \Gamma)$.

Proof: Correctness of the first point is trivial. The two following points have been proven previously by Broxvall and Jonsson [4] for the point algebra for partially ordered time. By examining these proofs it is obvious that they hold also for branching time since the reductions from NP-hard problems constructed in the proofs are satisfiable for partially ordered time iff they are satisfiable for branching time. For the fourth point consider the following set of relations:

$$a\ R'\ b\ ,\ b\ R'\ c\ ,\ c\ R'\ d\ ,\ x\ R\ y \vee a \| c\ ,\ z\ R\ w \vee b \| d$$

and note that at most one of $a \| c$ and $b \| d$ can hold. Hence, $x\ R\ y \vee z\ R\ w$ must hold without further restricting the choices of x, y, z, w. We use this in order to make a polynomial reduction from $\mathrm{SAT}_{\mathrm{br}}(\{\| \vee R, R', R \vee R\} \cup \Gamma)$ to $\mathrm{SAT}_{\mathrm{br}}(\{\| \vee R, R'\} \cup \Gamma)$. Let Π be an arbitrary problem instance of the first set of relations and replace each $R \vee R$ constraint of Π with the constraints above where a, b, c, d are fresh variables. Clearly the resulting set of constraints is an instance of $\mathrm{SAT}_{\mathrm{br}}(\{\| \vee R, R'\} \cup \Gamma)$ that is satisfiable iff the original instance is satisfiable. Correctness of the next two points follows by similar reasoning. The final point follows from the following two equivalences.

- $x\ \gamma_1\ y \vee z\ (\gamma_2 \cap \gamma_3)\ w$ is equivalent to $x\ \gamma_1\ y \vee z\ \gamma_2\ w$ and $x\ \gamma_1\ y \vee z\ \gamma_3\ w$
- $x\ \gamma_1\ y \vee z\ (\gamma_2 \circ \gamma_3)\ w$ is equivalent to $x\ \gamma_1\ y \vee z\ \gamma_2\ t$ and $x\ \gamma_1\ y \vee t\ \gamma_3\ w$ where t is a fresh variable.

□

We define the closure operation $\mathcal{C}(X)$ as the maximal set of relations constructible by successive applications of the previous rules. Note that $\mathcal{C}(X)$ is tractable iff X is tractable and $\mathcal{C}(X)$ is NP-complete iff X is NP-complete.

For the main result of this section, we use a construction which simplifies the proof by allowing us to consider only a finite number of disjunctions; this is shown in the next lemma proven by Broxvall and Jonsson [4].

Definition 4. *Let* $\mathcal{T} = \Gamma^{\veebar}\Delta^*$. *We define* $\overline{\mathcal{T}}$ *as* $(\mathrm{T} - \Gamma) \cup (\Gamma - \Delta)^{\veebar}(\Gamma - \Delta)$ *where* T *denotes the set of all binary relations composed by the union of basic relations from this domain.*

Lemma 8. *If* $\mathcal{T} = \Gamma^{\veebar}\Delta^*$, $\Delta \subseteq \Gamma$ *and* $\mathcal{T}' \not\subseteq \mathcal{T}$, *then there exists* $C \in \overline{\mathcal{T}}$ *such that* $C \in \mathcal{T}'$.

We can now prove the main result of this paper by using a computer assisted case analysis working in a similar way as the maximality proof of Broxvall and Jonsson [4].

Theorem 7. $\mathcal{T}_A, \mathcal{T}_B, \mathcal{T}_C, \mathcal{T}_D$ *and* $\mathcal{T}_E$ *are the only maximal tractable disjunctive subclasses of* $\mathrm{SAT}_{\mathrm{br}}$.

Proof: Suppose to the contrary that there exists another maximal tractable algebra $\mathcal{T}$. From the previous lemma it follows that there exists $\gamma_A, \cdots, \gamma_E$ in $\mathcal{T}$

such that $\gamma_A \in \overline{\mathcal{T}}_A, \cdots, \gamma_E \in \overline{\mathcal{T}}_E$. Note that there exists only a finite number of possible values for $\gamma_A, \ldots, \gamma_E$.

To prove the result, a machine-assisted[1] case analysis of the following form was performed: each admissible choice of $\gamma_A, \ldots, \gamma_E$ was generated and $\mathcal{T} = \mathcal{C}(\gamma_\mathrm{A}, \ldots, \gamma_\mathrm{E})$ was computed. Each such set $\mathcal{T}$ was examined and it was found that at least one of the NP-complete sets of Theorem 6 was a subset of $\mathcal{T}$. Thus, $\mathrm{SAT}_{\mathrm{br}}(\mathcal{T})$ is NP-complete and the theorem follows. □

6 Summary

We have identified five tractable sets of relations for the point algebra for branching time extended with disjunctions and by using a computer assisted case analysis shown these to be the only maximal tractable sets. We have also given a revised algorithm for determining satisfiability for the point algebra with a time complexity comparable to that of path consistency checking algorithms. Previously only algorithms running in $O(n^5)$ time have been known for this algebra.

References

1. J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
2. S. Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison Wesley, Reading, MA, 2nd edition, 1988.
3. M. Broxvall and P. Jonsson. Towards a complete classification of tractability in point algebras for nonlinear time. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 129–143, 1999.
4. M. Broxvall and P. Jonsson. Disjunctive temporal reasoning in partially ordered time structures. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 464–469. AAAI Press, 2000.
5. M. Broxvall, P. Jonsson, and J. Renz. Refinements and independence: A simple method for identifying tractable disjunctive constraints. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 114–127, 2000.
6. D. Cohen, P. Jeavons, P. Jonsson, and M. Koubarakis. Building tractable disjunctive constraints. *Journal of the ACM*, 47(5):826–853, 2000.
7. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Symbolic Computation*, 9(3):251–280, 1990.
8. I. Düntsch, H. Wang, and S. McCloskey. Relations algebras in qualitative spatial reasoning. *Fundamenta Informaticae*, 39(3):229–248, 1999.
9. E. A. Emerson and J. Srinivasan. Branching time temporal logic. In *Proceedings of REX Workshop 1988*, pages 123–172, 1988.
10. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.
11. R. Hirsch. Expressive power and complexity in algebraic logic. *Journal of Logic and Computation*, 7(3):309–351, 1997.

[1] The program is available online at `www.ida.liu.se/~matbr`

12. D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.
13. B. Nebel and H.-J. Bürckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. *Journal of the ACM*, 42(1):43–66, 1995.
14. J. Renz. Maximal tractable fragments of the region connection calculus: A complete analysis. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 129–143, Stockholm, Sweden, 1999.
15. M. B. Vilain, H. A. Kautz, and P. G. van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In D. S. Weld and J. de Kleer, editors, *Readings in Qualitative Reasoning about Physical Systems*, pages 373–381. Morgan Kaufmann, San Mateo, CA, 1989.

Exploiting Conditional Equivalences in Connection Calculi

Stefan Brüning

TLC GmbH, Weilburger Straße 28, D-60526 Frankfurt, Stefan.Bruening@tlc.de

Abstract. In a previous paper we proposed an approach to exploit literal equivalences in connection tableau based calculi. There we showed that making equivalences explicit offers new possibilities for search space reduction by applying literal demodulation for simplification and by strengthening the well-known regularity refinement. In this paper we generalize this approach to handle conditional equivalences. The generalization is mainly motivated by the circumstance that non-conditional equivalences, if not present at the beginning of a deduction, are much harder to generate than conditional ones.

1 Introduction

An inference mechanism working on a specific problem representation usually has no access to the information that is implicitly included in the representation of a formula. For instance, a formula in clausal form may implicitly contain information about equivalences that – if made explicit – can be used for simplification based on the fact that equivalence on literals is closely related to equality of terms. Given a literal-equivalence $A \equiv B$, any occurrence of literal B can be replaced by literal A in case $A < B$ holds for some Noetherian ordering. Such a use of equivalences provides some of the power of equality reasoning techniques for problems not noted in terms of equality, thus it partially solves a question stated by Wos: "What is the appropriate theory for demodulating across argument and across literal boundaries ... to replace certain predicates by other predicates and certain collections of literals by other collections?" [17].

In this vein, a calculus with logical equivalence was proposed in [16]. It combines resolution with the possibility to derive literal-equivalences and to use them as rewrite rules. It was shown that the reduction part of a corresponding calculus can be considerably improved by literal demodulation. In [6], a related approach was presented for the connection tableau calculus which is the basis for successful proof systems like Setheo [14] and KoMeT [4]. Since the connection tableau calculus is a top-down backward-chaining calculus – unlike saturation calculi based on resolution –, the situation is more complex because a straightforward application of demodulation would result in an incomplete calculus. However, it was shown that a careful exploitation of equivalences is feasible and can result in enormous reductions of the search space.[1] These reductions are due to simplifi-

[1] For instance, the resulting calculus is able to decide clause sets of the form $\{\{p(x), p(f(x))\}, \{\neg p(x), \neg p(f^n(x))\}\}$ within seconds. Without using equivalences

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 122–137, 2001.

cations achieved by demodulating the set of input clauses and by strengthening the regularity refinement. For generating equivalences dynamically in the course of a deduction, [6] also proposes a technique which is based on the analysis of non-regular tableaux (see Section 3 for more details).

The approaches presented in [16] and [6] share one basic restriction: Only the simplest form of equivalences – namely literal equivalences – is considered. Clearly, such equivalences are useful since they can be directly used for simplification. They are however – if not present at the beginning of a deduction – quite difficult to generate. Therefore, we propose an approach which is based on *conditional equivalences*, i.e. formulas of the form $C_1, \ldots, C_n \rightarrow A \equiv B$, which are, as we shall show in Section 3 comparatively easy to derive. To exploit conditional equivalences, we present two techniques: The first – and conceptual simpler – one uses lemmata (which are used in many proof systems for avoiding derivation duplication) to generate non-conditional equivalences from conditional ones. The second one aims at using conditional equivalences in situations where not enough information is available to turn them into non-conditional ones. We show that a conditional equivalence (of the above form) can be employed in a local fashion, that is, roughly speaking, it can be used for simplification during subderivations of some goal $\neg G$ in case it is guaranteed that in order to find an overall refutation, subrefutations for $\neg C_1, \ldots, \neg C_n$ have to be generated using a proof context that equals (or is smaller than) the one that is available for refuting $\neg G$.

The rest of the paper is organized as follows. Section 2 provides some basic terminology and summarizes the approach pursued in [6]. In Section 3 we demonstrate the difficulties in deriving non-conditional equivalences and present an approach to generate conditional equivalences. Sections 4 and 5 are devoted to the aforementioned techniques for exploiting conditional equivalences. In Section 6 we conclude.

2 Handling Non-conditional Equivalences

In this section we first provide some additional terminology concerning equivalences and show how literal-equivalences can be used for demodulation. Afterwards we introduce the basic concepts of the calculus presented in [6].

In what follows, we assume the reader to be familiar with the fundamental concepts of first-order clause logic. As usual, the variables occurring in clauses are considered implicitly as being universally quantified, a clause is considered logically as a disjunction of literals, and a clause set is taken as a conjunction of clauses. The letters u, v, w, x, y, z are used to denote variables, letters a, b, c are used to denote constant symbols. Let L^d denote the negation of a literal L, i.e. if L is an atom then $L^d = \neg L$, otherwise if $L = \neg A$ then $L^d = A$. L and L^d are called *complementary literals*. In addition to the standard definitions of atomic formulas and literals, we shall deal with atomic formulas that consist of an equivalence (we simply use the word equivalence instead of literal-equivalence).

this is a hard problem for top-down backward-chaining proof systems. SETHEO (version 3.0) [14] is not able to find a proof in less than one hour for $n = 20$.

Definition 1 (E-literal, Rule). *An* E-literal *is a pair* $\langle L, K\rangle$ *where* L *and* K *are literals. If the pair is ordered, then the E-literal* $\langle L, K\rangle$ *is also called a* rule*, and written in the form* $L \mapsto K$*. Otherwise, it is written in the form* $L \equiv K$*. We shall, however, still write* $\langle L, K\rangle$ *to include both possibilities* $L \equiv K$ *and* $L \mapsto K$*.*

For an E-literal $E = \langle L, K\rangle$*, we define* $Cl(E) = \{\{L, K^d\}, \{L^d, K\}\}$ *and for a set* $\mathcal{E}$ *of E-literals we set* $Cl(\mathcal{E}) = \bigcup_{E\in\mathcal{E}} Cl(E)$*. We sometimes use the term* equivalence *instead of* E-literal *if this cannot lead to confusion.*

Given a set of E-literals $\mathcal{E}$, $L \equiv_{\mathcal{E}} L'$ denotes that L is equivalent to L' wrt $\mathcal{E}$, i.e. there is a sequence $\langle L_1, K_1\rangle, \ldots, \langle L_n, K_n\rangle$ of elements of $\mathcal{E}$ and a substitution σ such that $L = L_1\sigma$, $K_i\sigma = L_{i+1}\sigma$ for all $1 \leq i < n$, and $K_n\sigma = L'$. In this case, we also say that L is $\mathcal{E}$*-equivalent* to L'. If $\mathcal{E} = \emptyset$, then $L \equiv_{\mathcal{E}} L'$ in case $L = L'$.

Using a reduction ordering, it is possible to direct E-literals to rules. A rule $R = L \mapsto K$ can be used to reduce literals in clauses as well as in E-literals: If for some substitution σ, $L\sigma = L'$, then R reduces L' to $K\sigma$. Since $L \equiv K$ is logically equivalent to $L^d \equiv K^d$, a rule used to reduce some literal L' is assumed to be in the form $L \mapsto K$ where L' and L have the same sign. An E-literal of the form $\langle L, L\rangle$ is called *trivial*. An E-literal E is subsumed by another E-literal E' if $E'\sigma = E$ for some substitution σ. Subsumed E-literals and trivial E-literals can always be removed.

In order to check a set of E-literals for unsatisfiability, so-called *L-paramodulation steps* are used. L-paramodulation steps allow to deduce new E-literals and correspond to a kind of superposition in the terminology of rewriting.

Definition 2 (L-Paramodulation Step). *Let* $E = \langle L, K\rangle$ *and* $E' = \langle L', K'\rangle$ *be E-literals. If there is a substitution* σ *such that* $L\sigma = L'\sigma$*, the E-literal* $\langle K\sigma, K'\sigma\rangle$ *is an* L-paramodulant *of* E *and* E'*. If there is a substitution* τ *such that* $L^d\tau = K\tau$*, then the empty clause is an* L-paramodulant *of* E*.*

In what follows, we introduce a connection tableau based calculus augmented by a refined handling of (non-conditional) equivalences. For brevity, we do not introduce the pure connection tableau calculus (e.g. see [12]) in a first step. Instead, we directly present the enhanced calculus given in [6]. Differences to the pure connection tableau calculus are explicitly pointed out.

The calculus presented in [6] does not only rely on connection tableaux as basic proof objects. Instead, a theorem to be proven is divided into a set of input clauses $\mathcal{S}$ and a set of E-literals $\mathcal{E}$ where we assume that $\mathcal{S}$ is irreducible wrt $\mathcal{E}$. The clauses in $\mathcal{S}$ are used for generating so-called connection E-tableaux using the derivation steps known from the pure connection tableau calculus, namely initialization, extension, and reduction steps. For the sake of completeness, an additional derivation step, called equivalence step, is required, which allows to modify E-tableaux using the elements of $\mathcal{E}$. Further, so-called L-Demodulation steps are used to reduce – during a deduction – the sets $\mathcal{S}$ and $\mathcal{E}$ as well as the actual E-tableau by dynamically generated equivalences (as we will see later, equivalences can be generated by inspecting non-regular E-tableaux).

Definition 3 (Connection E-Tableau). *A* clausal tableau *is a downward-oriented tree in which all nodes except the root node are labeled with literals.*

For every non-leaf node N in a clausal tableau, the sequence $N_1, \ldots, N_m$ of its immediate successor nodes is called the successor sequence *of N; if the nodes are labeled with literals $L_1, \ldots, L_m$, respectively, then the clause $\{L_1, \ldots, L_m\}$ is termed the* tableau clause below *N. The tableau clause below the root node is called* top-clause*.*

Let $\mathcal{S}$ be a clause set and $\mathcal{E}$ be a set of E-literals. A clausal tableau T is said to be an E-tableau *of $\mathcal{S}$ wrt $\mathcal{E}$ if for every tableau clause $\{L_1, \ldots, L_n\}$ in T, there is a substitution σ and a clause $\{K_1, \ldots, K_n\} \in \mathcal{S}$ such that $L_i \equiv_{\mathcal{E}} K_i\sigma$ $(1 \leq i \leq n)$. An E-tableau is called* connected *or* connection E-tableau *if each inner node labeled with a literal L has a leaf node among its immediate successors which is labeled with literal L^d.*

A branch *of a connected E-tableau T is a sequence $N_1, \ldots, N_n$ of nodes in T such that N_1 is the root node, N_i is the immediate predecessor of N_{i+1} $(1 \leq i < n)$, and N_n is a leaf. For $i \geq 1$, $N_i, \ldots, N_n$ is called a* subbranch starting with *N_i. A* (sub)branch *b* is closed *if it contains two nodes labeled with complementary literals. Otherwise, b is* open *and the literal attached to its leaf node is called an* open goal*. An* E-tableau is closed *if each of its branches is closed.*

The distinguishing feature of connection E-tableaux in contrast to ordinary connection tableaux (see [12]) is that a tableau clause needs not to be an instance of an input clause. Instead, it merely has to be equivalent to such an instance.

Definition 4 (Initialization Step). *At the beginning, select a clause from $\mathcal{S}$ and take it as top-clause of the E-tableau.*

Definition 5 (Extension Step). *Select a leaf node N of an open branch labeled with literal L. Let $\{L_1, \ldots, L_n\}$ be a new variant of a clause in $\mathcal{S}$ such that there exists an mgu σ with $L^d\sigma = L_i\sigma$ for some $1 \leq i \leq n$. Then, attach n new (immediate) successor nodes $N_1, \ldots, N_n$ to N, label them with $L_1, \ldots, L_n$, respectively, and apply σ to all tableau literals. We call N_i an* extension node*, and the elements of $\{N_1, \ldots, N_n\} - \{N_i\}$* non-extension nodes*. A literal attached to an extension node is called* extension literal*, otherwise* non-extension literal*. Furthermore, L_j is called the* original literal *of N_j for all $1 \leq j \leq n$.*

Definition 6 (Reduction Step). *Select a leaf node of an open branch labeled with literal L. If there is a node on the same branch labeled with literal L' such that there exists an mgu σ with $L^d\sigma = L'\sigma$, then apply σ to all tableau literals.*

A calculus which generates E-tableaux using initialization, extension, and reduction steps is sound and complete for first-order clause sets. In case E-literals are stored separately (in set $\mathcal{E}$), a further derivation step, called equivalence step, is required which allows to replace open goals by $\mathcal{E}$-equivalent literals.

Definition 7 (Equivalence Step). *Select a leaf node N of an open branch labeled with literal L. If there is a set $\{\langle L_1, K_1\rangle, \ldots, \langle L_n, K_n\rangle\}$ of variants of elements of $\mathcal{E}$ and an mgu σ such that $L\sigma = L_1\sigma$ and $K_i\sigma = L_{i+1}\sigma$ for all $1 \leq i < n$, then apply σ to all tableau literals and label N with $K_n\sigma$.*

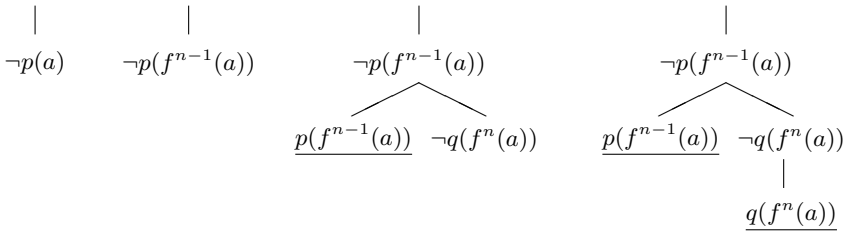


Fig. 1. E-tableaux for Example 1. Leaf nodes of closed branches are underlined.

Example 1. For illustration, consider the following clause set $\mathcal{S}$.[2]

$$\begin{array}{lll} (1)\ q(f^n(a)) \leftarrow & (2)\ q(v) \leftarrow p(v) & (3)\ p(u) \leftarrow q(f(u)) \\ (4)\ p(y) \leftarrow p(f(y)) & (5)\ p(f(z)) \leftarrow p(z) & (6) \leftarrow p(a) \end{array}$$

Clauses (4) and (5) encode the E-literal $E = \langle p(f(y)), p(y)\rangle$ which can be directed to a rule $R = p(f(y)) \mapsto p(y)$. Rewriting $\mathcal{S}$ with R, clauses (4) and (5) become tautological and are removed. Now consider the E-tableaux in Figure 1. The first one is built by an initialization step with clause (6), the second one results from an equivalence step with $n-1$ instances of E. An extension step with clause (3) generates the third E-tableau, the last (closed) E-tableau is generated by a further extension step with clause (1).

Remark 1. The need for equivalence steps is due to the connectedness criterion given in Definition 3. One might object that this makes a refined handling of equivalences useless. This is fortunately not the case since the search for 'minimal' refutations by demodulating the set of input clauses may reduce the search space considerably. For more details and illustration, see [6].

In view of an implementation one should take into consideration that the application of one equivalence step encodes several implicit extension steps using clauses from $Cl(\mathcal{E})$. The number of these implicit steps has to be taken into account if some kind of iterative-deepening search strategy is performed.

Definition 8 (E-Derivation, E-Refutation). *A sequence D of initialization, extension, reduction, and equivalence steps generating a connection E-tableau T from a clause set $\mathcal{S}$ and a set of E-literals $\mathcal{E}$ is called an* E-derivation for $\mathcal{S}$ and $\mathcal{E}$. *If T is closed, D is called* E-refutation.

Let L be an open goal attached to a node N in a connection E-tableau T. A subderivation *D for N (or L) is an E-derivation where the first element of D selects N and each further element selects a descendant of N. D is called a* subrefutation *if after applying D to T, each branch containing N is closed.*

[2] In many cases we write clauses in a PROLOG like manner.

The number of connection tableau derivations to be taken into account can be restricted considerably by using the *regularity restriction* (e.g. see [14]) which allows to ignore E-tableaux containing two identical literals on one branch. Interestingly, the use of equivalences allows to strengthen the regularity restriction. Instead of requiring two identical literals on one branch to be applicable, the literals only have to be logically equivalent.

Definition 9 (E-Regular E-Tableau). *Let $\mathcal{E}$ be a set of E-literals. An E-tableau T is* E-regular *wrt $\mathcal{E}$ (or $\mathcal{E}$-*regular*) if no two different nodes on a branch are labeled with literals that are $\mathcal{E}$-equivalent.*

Remark 2. In case $\mathcal{E} = \emptyset$, E-regularity equals "classical" regularity. Since it is in general undecidable whether two literals are equivalent, one should – in order to keep the E-regularity check simple – only test whether a branch contains two literals that can be *reduced* to the same literal. For more details, see [6].

Example 2. We continue Example 1. After an initialization step with clause (6) two extension steps with clauses (3) and (2), respectively, are applied. The resulting E-tableau has one open branch which contains the literals $\neg p(a)$ and $\neg p(f(a))$. Thus, the E-tableau is not E-regular wrt $\{\langle p(f(y)), p(y)\rangle\}$, and therefore the derivation is pruned. In the same way, any E-derivation applying extension steps using clauses (2) and (3) to generate the term $f^n(a)$ is pruned. Therefore, the only possibility to derive a closed E-tableau is to apply an equivalence step (as in Example 1). This reduces a search space of exponential size (in n) to one of linear size. Note, that in case only the "classical" regularity restriction is employed, the size of the pruned search space remains exponential.

The following theorem is the main result of [6].

Theorem 1. *Given a clause set $\mathcal{S}$ and a set of E-literals $\mathcal{E}$, $\mathcal{S} \cup Cl(\mathcal{E})$ is unsatisfiable iff there exists an E-refutation for $\mathcal{S}$ and $\mathcal{E}$ generating a closed $\mathcal{E}$-regular E-tableau, or the empty clause is derivable from $\mathcal{E}$ by L-paramodulation steps.*

So far, we have not discussed the possibility to exploit equivalences which are generated *during* a deduction (see Section 3). If a set of new E-literals is derived, the current set of E-literals as well as the set of input clauses can be reduced. As a consequence, the current E-tableau T may not be any longer an E-tableau of the reduced clause set. Then, some of the derivation steps generating T have to be withdrawn. This is accomplished by a so-called *L-demodulation step.*

Definition 10 (L-Demodulation Step). *Let $\mathcal{S}$ be a set of clauses, $\mathcal{E}$ be a set of E-literals, and T be a connection E-tableau of $\mathcal{S}$ wrt $\mathcal{E}$ generated by an E-derivation $d_1, \ldots, d_n$. Let $\mathcal{E}'$ be a set of E-literals that is logically implied by $\mathcal{S} \cup Cl(\mathcal{E})$. An* L-demodulation step *consists of three successive substeps:*

(i) Reduce $\mathcal{E} \cup \mathcal{E}'$ to an irreducible set $\mathcal{E}''$ (that is no rule in $\mathcal{E}''$ can be used to reduce another E-literal in $\mathcal{E}''$). (ii) Reduce $\mathcal{S}$ wrt $\mathcal{E}''$ to an irreducible set $\mathcal{S}''$. (iii) Take derivation steps back (starting with d_n) until the resulting E-tableau is an E-tableau of $\mathcal{S}''$ wrt $\mathcal{E}''$.

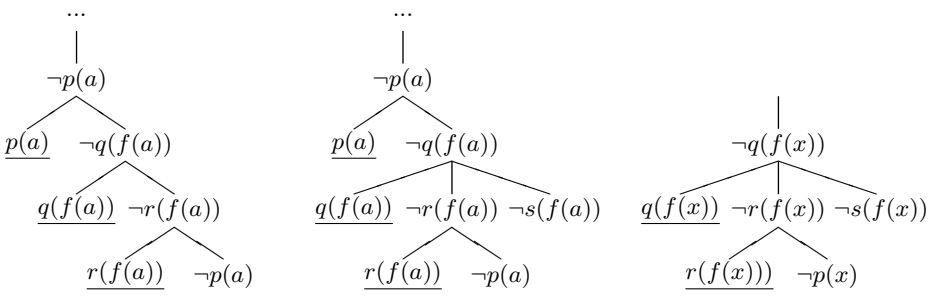


Fig. 2. E-Tableaux for Examples 3, 4, 5 and 6. Leaf nodes of closed branches are underlined.

After the application of an L-demodulation step, the deduction process continues with the sets $\mathcal{S}''$ and $\mathcal{E}''$. Note that an L-demodulation step is a simplification step. It has not to be taken back during the entire deduction. In [6] it is shown that L-demodulation steps preserve correctness and completeness.

3 Generation of Conditional Equivalences

Besides providing a calculus for exploiting (non-conditional) equivalences, [6] also contains an approach for generating new equivalences during a deduction. This is useful in those cases where equivalences are only included implicitly in the (clausal) representation of a problem. The basic idea of [6] is to generate new equivalences from E-tableaux which are not E-regular.

Example 3. Consider a clause set $\mathcal{S}$ containing (among others) the clauses

$$(1)\ p(x) \leftarrow q(f(x)) \qquad (2)\ q(y) \leftarrow r(y) \qquad (3)\ r(f(z)) \leftarrow p(z)$$

and an E-derivation for $\mathcal{S}$ generating the open goal $\neg p(a)$. Applying three extension steps using clauses (1), (2), and (3), respectively, results in a non-regular E-tableau T (the first one depicted in Figure 2). This E-tableau makes the cycle of implications included in $\mathcal{S}$ explicit what allows to derive the E-literals $\langle p(x), q(f(x))\rangle$, $\langle q(f(x)), r(f(x))\rangle$, and $\langle p(x), r(f(x))\rangle$ (note that the substitution of the variables in T by constant a is due to the initial open goal $\neg p(a)$).

However, this approach has one – quite severe – restriction: Let $N_1, \ldots, N_n$ be an open subbranch where N_1 and N_n are labeled with equivalent literals. Then, in order to generate new E-literals from the corresponding E-tableau, *each* subbranch starting in N_1 except the one ending in N_n must be closed.[3]

[3] An in its essence similar restriction holds for the resolution based calculus presented in [16]. There, in order to generate a literal equivalence two binary clauses have to be resolved. This requires the previous derivation of "suitable" binary clauses.

Example 4. We continue Example 3 and replace clause (2) with clause (2′) $q(y) \leftarrow r(y), s(y)$. As before, we can generate a non-regular E-tableau by applying three extension steps using clauses (1), (2′), and (3), respectively. The resulting E-tableau T' is the second one depicted in Figure 2. Since T' contains the open goal $\neg s(f(a))$, it is no longer possible to generate E-literals.

Remark 3. One might argue that the situation given in the previous example changes if another goal selection strategy is used throughout the deduction. Then $\neg s(f(a))$ might be selected before $\neg r(f(a))$ which would allow to derive E-literals after generating a subrefutation for $\neg s(f(a))$. It is very likely, however, that such a different selection strategy applies 'wrong' selections in other situations.

The approach pursued in this paper is to overcome this problem by deriving *conditional equivalences*, that is, formulas of the form $C_1 \wedge \ldots \wedge C_n \rightarrow \langle A, B\rangle$ where, roughly speaking, $C_1, \ldots, C_n$ correspond to (the negations) of the open goals that prevent the generation of non-conditional equivalences. Reconsidering Example 4 where we have one such open goal, namely $\neg s(f(a))$, it should be possible to generate $s(f(x)) \rightarrow \langle p(x), q(f(x))\rangle$, $s(f(x)) \rightarrow \langle q(f(x)), r(f(x))\rangle$, and $s(f(x)) \rightarrow \langle p(x), r(f(x))\rangle$ (note again that the substitution of the variables in T' by constant a is due to the initial open goal $\neg p(a)$).

In what follows, we use the following terminology: A formula E of the form $C_1 \wedge \ldots \wedge C_n \rightarrow \langle A, B\rangle$ is called *conditional E-literal* if $n > 0$. In accordance with Definition 1, we use the terms conditional E-literal and conditional equivalence interchangeably if no confusion is possible. If $n = 0$, we often call E a *non-conditional equivalence* (instead of equivalence or E-literal). $C_1, \ldots, C_n$ are called the *condition literals* of E. E is called *trivial* if $A = B$. E is *subsumed* by an E-literal E' if there is a substitution σ such that $E'\sigma = \langle A, B\rangle$.

To formalize the generation of new conditional E-literals from E-tableaux that are not E-regular, we use so-called *subtableaux*. Basically, a subtableau of an E-tableau T is a subtree of T. We use subtableaux to isolate those parts of a derivation which can be used for the derivation of (conditional) equivalences. Importantly, the overall substitution required to build a subtableau of T is usually more general than the overall substitution required for the generation of T. This gives us the possibility to derive more general (conditional) equivalences than by inspecting the literals of T directly.

Definition 11 (Subtableau). *Let T be an E-tableau generated by an E-derivation D and let $N_1, \ldots, N_n$ be the non-extension nodes of a successor sequence in T. The* subtableau of T rooted with $N_1, \ldots, N_n$ *is generated as follows:*

First, an initialization step with top-clause $\{L_1, \ldots, L_n\}$ is performed, where L_i is the original literal (see Definition 5) of N_i $(1 \leq i \leq n)$. The resulting E-tableau contains n leaf nodes $M_1, \ldots, M_n$ labeled with $L_1, \ldots, L_n$, respectively. Afterwards, the extension and equivalence steps in D that where applied to N_i (and its successors) are applied to M_i (with possibly different mgus), for all $1 \leq i \leq n$. A reduction step is carried over only in case no ancestor of N_i was used to apply the reduction step.

Example 5. We continue Example 4. Let N denote the node in T' labeled with literal $\neg q(f(a))$. The subtableau T'' of T' rooted with N is depicted in Figure 2 on the right. Note, that the variables in T'' are not bound to constant a.

Theorem 2. *Let T_1 be an E-tableau of a clause set $\mathcal{S}$ wrt a set of E-literals $\mathcal{E}$, and $N_1, \ldots, N_n$ be some open subbranch of T_1. Let $U_1, \ldots, U_m$ be the successor sequence of N_1 with U_1 being an extension node. Further, let T_2 be the subtableau of T_1 rooted with $U_2, \ldots, U_m$ and σ be the overall substitution of the derivation generating T_2. If the literals attached to N_1 and N_n are $\mathcal{E}$-equivalent, then*

$$\mathcal{S} \cup Cl(\mathcal{E}) \vdash C_1^d\tau \wedge \ldots \wedge C_l^d\tau \rightarrow L_2\tau \equiv_{\mathcal{E}} L_3\tau \wedge \ldots \wedge L_{n-1}\tau \equiv_{\mathcal{E}} L_n\tau,$$

where (i) for $2 \leq i \leq n$, L_i is the literal attached to the node in T_2 which corresponds to node N_i in T_1, (ii) $C_1, \ldots, C_l$ are the open goals in T_2 except L_n, and (iii) τ is a substitution such that $L_1^d\sigma\tau \equiv_{\mathcal{E}} L_n\tau$ where L_1 denotes the original literal of U_1.

Note that the label L of U_1 in T_1 is the complement of the literal attached to N_1 in T_1 and that L is an instance of L_1 since no equivalence step can be applied to extension nodes.

In case the literals attached to N_1 and N_n in T_1 – say L_1' and L_n' – are equal, we obtain τ as the mgu of $L_1^d\sigma$ and L_n. Otherwise, the equivalence of L_1' and L_n' can be shown via a sequence of E-literals $\langle K_1, K_1' \rangle, \ldots, \langle K_r, K_r' \rangle$ (see Section 2). Then, τ is the mgu such that $L_1^d\sigma\tau = K_1\rho\tau$ and $K_r'\rho\tau = L_n\tau$ where ρ is the mgu which unifies K_i' and K_{i+1} for $1 \leq i < r$.

Remark 4. There are two quite obvious generalizations of Theorem 2. The first one is to demand that N_1 and some N_j with $j \leq n$ (instead of N_1 and N_n) are labeled with equivalent literals and to generate (conditional) equivalences from the literals attached to $N_1, \ldots, N_j$. The second one is to take an additional substitution γ into account and to generate (conditional) equivalences from E-tableaux where the labels of N_1 and N_j are $\mathcal{E}$-equivalent *after* the application of γ. For the sake of simplicity, however, we will stick to Theorem 2; the first generalization would, for instance, require a generalization of Definition 11.

Example 6. We continue Example 5. In terms of Theorem 2 we have: T_1 and T_2 are the second and third E-tableau depicted in Figure 2, respectively, $\sigma = \{y \backslash f(x), z \backslash x\}$, and $\tau = \emptyset$. Following Theorem 2 we get

$$\mathcal{S} \cup Cl(\mathcal{E}) \vdash s(f(x)) \rightarrow p(x) \equiv q(f(x)) \ \wedge \ q(f(x)) \equiv r(f(x)) \ \wedge \ p(x) \equiv r(f(x))$$

what results in the conditional E-literals $s(f(x)) \rightarrow \langle p(x), q(f(x)) \rangle$, $s(f(x)) \rightarrow \langle q(f(x)), r(f(x)) \rangle$, and $s(f(x)) \rightarrow \langle p(x), r(f(x)) \rangle$.

The exhaustive use of Theorem 2 might lead to the generation of a huge number of conditional E-literals.[4] Therefore, we recommend the usage of heuristics to

[4] Similar problems occur in case lemmata are generated in a non-restricted manner, see [1,13].

reduce this number. One such heuristic is to avoid conditional E-literals with more than one or two condition literals. In case non-ground derivations (that generate tableaux containing many non-ground open goals) are considered, another heuristic is to ignore ground conditional E-literals since their applicability is limited in such cases.

4 Lemma Handling and Conditional Equivalences

Lemma handling (e.g. see [1,13,10]) is a basic technique to avoid derivation duplication in top-down backward-chaining calculi, like model elimination or connection tableau calculi. The simplest – albeit most useful – form of lemmata are so-called *unit lemmata* that consist of only one literal. Roughly speaking, they can be derived in case an E-tableau contains a closed subtableau. Given a unit lemma L, which is generated during an E-derivation for a clause set $\mathcal{S}$ and a set of E-literals $\mathcal{E}$, it is guaranteed that $\mathcal{S} \cup Cl(\mathcal{E}) \vdash L$ holds.[5]

In this section we show that besides using unit lemmata for avoiding derivation duplication, they are also useful for generating non-conditional equivalences from conditional ones. Since most proof systems based on the connection tableau calculus (e.g. Setheo [14,13] or KoMeT [4]) already employ lemmata such a combination is quite attractive.

In what follows, the set of conditional E-literals generated throughout a deduction is denoted by $\mathcal{CE}$, the set of derived unit lemmata is denoted by $\mathcal{L}$. We assume that $\mathcal{CE}$ as well as $\mathcal{L}$ are reduced wrt $\mathcal{E}$. Basically, a non-conditional equivalence (i.e. an E-literal) can be obtained from a conditional E-literal in case each of the condition literals (or a more general literal) has been proved as unit lemma or is already contained as unit clause in the set of input clauses.

Theorem 3. *Let $\mathcal{S}$ be a clause set, $\mathcal{E}$ be a set of E-literals, $\mathcal{CE}$ be a set of conditional E-literals, and $\mathcal{L}$ be a set of unit lemmata such that the elements of $\mathcal{CE}$ and $\mathcal{L}$ are logical implications of $\mathcal{S} \cup Cl(\mathcal{E})$.*

Let $C_1 \wedge \ldots \wedge C_n \rightarrow \langle A, B \rangle$ be an element of $\mathcal{CE}$ and $L_1, \ldots, L_n$ be a sequence of literals such that for $1 \leq i \leq n$, $L_i \in \mathcal{L}$ or $\{L_i\} \in \mathcal{S}$ holds. If there is a substitution σ such that $C_i\sigma = L_i\sigma$ for $1 \leq i \leq n$, then $\mathcal{S} \cup Cl(\mathcal{E}) \vdash A\sigma \equiv B\sigma$.

An integration of this approach for handling conditional equivalences into the calculus presented in Section 2 requires the following extensions:

- An initialization step has to initialize the sets $\mathcal{CE}$ and $\mathcal{L}$ to the empty set.
- Unit lemmata and conditional E-literals, that are generated throughout the deduction have to be reduced wrt $\mathcal{E}$ and added to $\mathcal{L}$ and $\mathcal{CE}$, respectively.
- Definition 10 (L-demodulation step) has to be extended as follows: (iv) The sets $\mathcal{CE}$ and $\mathcal{L}$ have to be reduced wrt $\mathcal{E}''$ to sets $\mathcal{CE}''$ and $\mathcal{L}''$, respectively, and (v) elements of $\mathcal{CE}''$ have to be deleted if they are trivial or subsumed by elements of $\mathcal{E}''$.

[5] There is no difference for the generation of lemmata between "ordinary" connection tableau derivations and E-derivations as defined in this paper.

Note that in case an E-literal $\langle A\sigma, B\sigma\rangle$ is generated according to Theorem 3, the corresponding conditional E-literal is deleted by a following L-demodulation step from $\mathcal{CE}$ only in case $\langle A\sigma, B\sigma\rangle$ and $\langle A, B\rangle$ are equal up to variable renaming.

Example 7. We continue Example 6. As we have seen, we can add the conditional E-literals $s(f(x)) \rightarrow \langle p(x), q(f(x))\rangle$, $s(f(x)) \rightarrow \langle q(f(x)), r(f(x))\rangle$, and $s(f(x)) \rightarrow \langle p(x), r(f(x))\rangle$ to $\mathcal{CE}$. In case is it possible to derive unit lemma $s(f(a))$ afterwards, Theorem 3 allows us to generate the set of E-literals $\mathcal{E}'_1 = \{\langle p(a), q(f(a))\rangle, \langle q(f(a)), r(f(a))\rangle, \langle p(a), r(f(a))\rangle\}$. More useful, i.e. more general E-literals can be generated in case a more general lemma, for instance $s(y)$, can be deduced. Then, we get $\mathcal{E}'_2 = \{\langle p(x), q(f(x))\rangle, \langle q(f(x)), r(f(x))\rangle, \langle p(x), r(f(x))\rangle\}$. Reducing $\mathcal{CE}$ wrt $\mathcal{E}'_2$ (what happens by the application of an L-demodulation step), the previously generated conditional E-literals become trivial and can be removed.

Remark 5. One might object that in case unit lemmata are employed, there is no need for handling conditional equivalences at all. For illustration reconsider Example 6. If it is possible to deduce a unit lemma of the form $s(f(...))$ or $s(y)$, one might argue that it should be possible to derive (non-conditional) E-literals directly. However, this is only true if such a unit lemma is generated *before* the second E-tableau (see Figure 2). Otherwise, in case no conditional E-literals are generated, the possibility to derive non-conditional ones might be lost since the same situation might not occur again (after the generation of the unit lemma). This holds even if backtracking is taken into account since the applicability of lemmata might prevent subderivations that are generated if lemmata are not present. For instance, in case lemma $p(a)$ is available to close a branch with open goal $\neg p(a)$, there is no need to consider alternative subderivations for $\neg p(a)$.

Another argument in favor of handling conditional equivalences is the possibility to derive different non-conditional E-literals from one conditional E-literal (depending on the applied unit lemmata as illustrated in Example 7). In [6], generated E-literals only depend on the respective tableaux that violate E-regularity. Reconsidering Example 7, this may lead to a situation where only the set of E-literals $\mathcal{E}'_1$ can be derived instead of the more useful set $\mathcal{E}'_2$.

5 Using Conditional Equivalences for Local Simplification

In this section we present an approach that aims at exploiting conditional equivalences in situations where not enough information – like unit lemmata – is available to generate non-conditional equivalences. Roughly speaking this approach is based on the idea that given some conditional E-literal $C_1 \wedge \ldots \wedge C_n \rightarrow \langle L, K\rangle$, E-literal $\langle L, K\rangle$ can be used if it is guaranteed that in order to find an overall refutation, subrefutations for $C_1^d, \ldots, C_n^d$ must exist, too. For preserving soundness, it is essential to apply $\langle L, K\rangle$ in a local fashion: If $\langle L, K\rangle$ is applied during a subderivation of some goal G, it must be guaranteed that G has at least the same *preconditions* (i.e. ancestor literals for the application of reduction steps) that are available to find subrefutations for $C_1^d, \ldots, C_n^d$.

Definition 12 (Precondition). *Let L be a literal attached to an node N in a connection E-tableau. The set of literals attached to the ancestors of N is called the* precondition *of L and denoted by Pre_L.*

The precise conditions that have to be met in order to use a conditional E-literal like a non-conditional one are given in the following definition:

Definition 13 (Justification for Conditional E-Literals). *Let G be an open goal in a connection E-tableau T and let $E = C_1 \wedge \ldots \wedge C_n \rightarrow \langle L, K \rangle$ be a conditional E-literal. If for some substitution σ, T contains besides G non-extension literals $C_1^d\sigma, \ldots, C_n^d\sigma$, such that (i) for $1 \leq i \leq n$, $Pre_{C_i^d\sigma} \subseteq Pre_G$, and (ii) $var(C_1\sigma \wedge \ldots \wedge C_n\sigma) \cap var(\langle L\sigma, K\sigma \rangle) = \emptyset$[6], then G has a* justification *for $E\sigma$.*

Condition (ii) of the above Definition assures, that substitutions which may be required to complete the subrefutations of $C_1^d\sigma, \ldots, C_n^d\sigma$ do not affect the applicability of $\langle L\sigma, K\sigma \rangle$ during subderivations of G (note that $C_1^d\sigma, \ldots, C_n^d\sigma$ may be *inner literals*, they do not have to be open goals). Without this condition it might be the case that $\langle L\sigma, K\sigma \rangle$ is used for simplification although the subrefutations for $C_1^d\sigma, \ldots, C_n^d\sigma$ require an additional substitution τ that only allows the usage of an instance of $\langle L\sigma, K\sigma \rangle$ (namely $\langle L\sigma\tau, K\sigma\tau \rangle$).

Theorem 4. *Let $E = C_1 \wedge \ldots \wedge C_n \rightarrow \langle L, K \rangle$ be a conditional E-literal and G be an open goal which has a justification for E. Then, $\langle L, K \rangle$ can be used throughout subderivations of G (without sacrificing soundness or completeness).*

Remark 6. Obviously, the usefulness of the above theorem increases with the number of ground non-extension literals that are generated during a deduction. However, even if a derivation generates lots of non-ground open goals, it is often the case that these non-ground goals become ground by the application of derivation steps. Further, many application domains consider restricted classes of clause sets that only allow derivations where all – or at least most – open goals (and therefore also all – or at least most – (inner) non-extension literals) are ground. For instance, this is the case for (some kind of) range-restricted clause sets that are used by connection-tableau based calculi for default reasoning [7, 8] or in the area of deductive databases [9].

The application of Theorem 4 requires some form of "local" L-demodulation steps: In subderivations of G it is possible to use $\langle L, K \rangle$ for simplification and equivalence steps. Outside subderivations for G this might be impossible since the respective open goals might not have a justification for E. Hence, the notion of E-tableaux (Definition 3) has to be generalized: An E-tableau cannot be defined any longer wrt to a fixed set of input clauses. Instead, it must be allowed to contain tableau clauses that are equivalent to instances of input clauses which are reduced wrt E-literals that can only be applied locally. Due to lack of space we have to skip the corresponding formal definitions. The following example illustrates our approach.

[6] By $var(E)$ we denote the set of variables occurring in E.

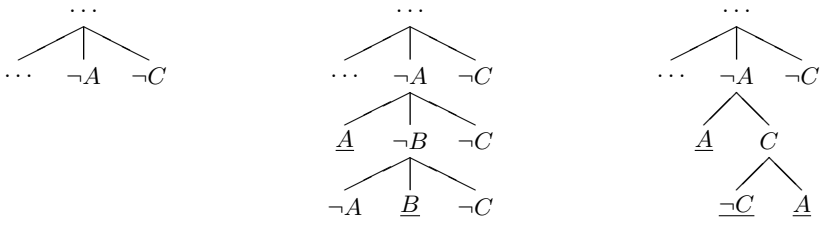


Fig. 3. Tableaux for Examples 8 and 9. Leaf nodes of closed branches are underlined.

Example 8. Let $\mathcal{S}$ be a set containing (among others) the following clauses:

(1) $\{A, B, C\}$ (2) $\{A, B, \neg C\}$ (3) $\{A, \neg B, C\}$ (4) $\{A, \neg B, \neg C\}$
(5) $\{\neg A, B, C\}$ (6) $\{\neg A, B, \neg C\}$ (7) $\{\neg A, \neg B, C\}$

Consider an E-tableau of $\mathcal{S}$ containing a tableau clause cl with open goals $\neg A$ and $\neg C$ as depicted in Figure 3 on the left. Applying two extension steps with clauses (4) and (6), the second (non-regular) E-tableau depicted in Figure 3 can be generated which allows to derive the conditional E-literal $E = C \rightarrow \langle A, B\rangle$.

Since literal $\neg A$ contained in tableau clause cl has the same precondition than literal $\neg C$ (in cl), $\neg A$ has a justification for E. Hence we can use E-literal $E' = \langle A, B\rangle$ during subderivations for $\neg A$. Assuming that $A < B$ for some reduction ordering $<$ holds, we direct E' to a rule $B \mid\!\!\rightarrow A$ and simplify $\mathcal{S}$. The resulting set contains instead of clauses (1) – (7) the following ones:

$$(1')\ \{A, C\} \qquad (2')\ \{A, \neg C\} \qquad (7')\ \{\neg A, C\}$$

Now it is possible to find a short subrefutation for $\neg A$ consisting of two extension steps with clauses (1') and (2') (see the third E-tableau in Figure 3). A subrefutation that makes no use of equivalences needs at least 5 extension steps.

It should be noticed that the technique presented in Section 4 might not be applicable in the given situation, even if another goal selection strategy is used: Since a subrefutation for $\neg C$ might depend on ancestor literals of $\neg C$, it might be impossible to find a subrefutation for $\neg C$ that allows the derivation of unit lemma C.

The next example shows that in order to use a conditional E-literal E during subderivations of some goal G, it is necessary that G has a justification for E.

Example 9. We continue Example 8 and replace clause (5) by clause $\{\neg A, B, D\}$. As before, it is possible to generate the conditional E-literal $E = C \rightarrow \langle A, B\rangle$ and to find a short subrefutation for $\neg A$. On the other hand, it is easy to verify that no classical subrefutation (i.e. a subrefutation that only uses extension and reduction steps) for the remaining open goal $\neg C$ can be found. Now, assume that $\langle A, B\rangle$ can be used safely during subderivations of $\neg C$ (although $\neg C$ has no justification for E). Then, for the sake of soundness, it should not be possible to find a subrefutation for $\neg C$ using clauses (1'), (2'), and (7'). This however is

possible by applying two extension steps with clauses (7') and (1'), respectively. Hence, the application of $\langle A, B\rangle$ during derivations of $\neg C$ is unsound.

Similar problems occur if a combination of the techniques presented in this and the previous section is considered. If both are combined, one has to guarantee that in case a (conditional) E-literal E (or a lemma L) is generated from a (sub)derivation that makes use of some conditional E-literal E' (according to Theorem 4), then E (or L) must not be used during subderivations of goals that have no justification for E'.

Finally, we mention that conditional equivalences can also be exploited for a refined handling of E-regularity: Whenever some open goal G has a justification for a conditional E-literal $C_1 \wedge \ldots \wedge C_n \rightarrow \langle A, B\rangle$, no subbranch starting in G is allowed to contain literals that are equivalent wrt to $\mathcal{E} \cup \{\langle A, B\rangle\}$ where – as above – $\mathcal{E}$ denotes the set of E-literals used throughout the deduction.

6 Conclusion

We proposed an approach for handling conditional equivalences in connection calculi. It extends previous work, presented in [6], where it was shown that a combination of connection calculi with the ability to exploit literal equivalences for simplification has a great potential to reduce the search space. The presented extensions are, on the one hand, a method for deriving conditional equivalences and, on the other hand, two mechanisms for exploiting conditional equivalences. The first of these techniques allows to turn conditional equivalences into non-conditional ones by a combination with lemma handling. The second one shows how conditional equivalences can be used directly in a local fashion for search space reduction. We argued that the exploitation of conditional equivalences is more promising in comparison to the approach presented in [6] since conditional equivalences are easier to generate than non-conditional equivalences. The aforementioned second technique further shows that conditional equivalences can be useful in situations where it is not possible to derive non-conditional ones.

We do not claim that the use of (conditional) equivalences makes it possible to find shorter proofs for any class of clause sets. In those cases where no equivalences are included (explicitly or implicitly) in the problem representation, the presented mechanisms will be of no help. Even worse, their application will reduce the inference rate of the proof system. In other cases, however, the exploitation of equivalences allows to find proofs which are otherwise unobtainable in reasonable time (for instance see Example 2 or Footnote 1). Hence, we recommend to implement the proposed techniques as an optional feature which can be switched on or off. Many calculi refinements – like lemma handling – are usually implemented in this way (since no technique is useful in all cases).

The results of this paper should be equally applicable to other top-down backward-chaining connection calculi, as they are presented in [3]. Even an extension of the resolution based approach in [16] seems to be quite straightforward, with one exception: There is no direct way to carry over the technique presented

in Section 5 since it makes use of the proof structure given by connection E-tableaux (see Definitions 12 and 13).

An interesting direction to generalize our results is to allow arbitrary equivalences. For instance, it would be quite attractive to handle equivalences of the form $bachelor(x) \equiv male(x) \wedge \neg married(x)$ for problems in the area of Knowledge Representation (e.g. see [5]). Approaches that use arbitrary equivalences (in combination with calculi that differ significantly from the connection tableau calculus) are presented in [11] and [15]. In both cases, however, completeness of the resulting calculi and the possibility to derive new equivalences are not considered. In order to extend the approach pursued in this paper, it seems to be necessary to take formulas in non-normal form (instead of clausal form) into account (e.g. see [3]). Another direction for future research is the integration of equivalences in *confluent* connection calculi (e.g. see [2]). In such calculi it should be possible to achieve completeness without equivalence steps. This would clearly enhance the effectiveness of using equivalences for demodulation.

References

1. O. L. Astrachan and M. E. Stickel. Caching and Lemmaizing in Model Elimination Theorem Provers. In *Proc. of CADE*, vol. 607 of *LNAI*, pages 224–238. Springer, 1992.
2. P. Baumgartner, N. Eisinger, and U. Furbach. *Intellectics and Computational Logic*, chapter A Confluent Connection Calculus, pages 3–26. Kluwer, 2000.
3. W. Bibel. *Deduction: Automated Logic*. Academic Press, London, 1993.
4. W. Bibel, S. Brüning, U. Egly, and T. Rath. Komet. In *Proc. of CADE*, number 814 in LNAI, pages 783–787. Springer, 1994. System description.
5. G. Brewka, editor. *Principles of Knowledge Representation*. CSLI Publications, 1996.
6. S. Brüning. Exploiting Equivalences in Connection Calculi. *Journal of the Interest Group in Pure and Applied Logics (IGPL)*, 3(6):857–886, 1995.
7. S. Brüning and T. Schaub. Prolog technology for default reasoning: Proof theory and compilation techniques. *Artificial Intelligence*, 106(1):1–75, 1998.
8. S. Brüning and T. Schaub. Avoiding non-ground variables. In *Proc. of the Fifth European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, vol. 1638 of *LNAI*, pages 92–103. Springer, 1999.
9. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990.
10. M. Fuchs. System Description: Similarity-Based Lemma Generation for Model Elimination. In *Proc. of CADE*, vol. 1421 of *LNAI*, pages 33–37. Springer, 1998.
11. S.-J. Lee and D. A. Plaisted. Reasoning with Predicate Replacement. In *Proc. of ISMIS*, 1990.
12. R. Letz. *First-Order Calculi and Proof Procedures for Automated Deduction*. PhD thesis, TH Darmstadt, 1993.
13. R. Letz, K. Mayr, and Ch. Goller. Controlled Integrations of the Cut Rule into Connection Tableau Calculi. *Journal of Automated Reasoning*, 13(3):297–338, 1994.
14. R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO — A High–Performance Theorem Prover for First–Order Logic. *Journal of Automated Reasoning*, 8:183–212, 1992.

15. A. F. McMichael. An Automated Reasoner For Equivalences, Applied To Set Theory. In *Proc. of CADE*, pages 308–321, 1990.
16. R. Socher-Ambrosius. A Resolution Calculus Extended by Equivalence. In *Proc. of the German Workshop on Artificial Intelligence*, pages 102–106. Springer, 1989.
17. L. Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice Hall, Englewood Cliffs, 1988.

Propositional Satisfiability in Answer-Set Programming

Deborah East and Mirosław Truszczyński

Department of Computer Science, University of Kentucky
Lexington KY 40506-0046, USA
{deast, mirek}@cs.engr.uky.edu

Abstract. We show that propositional logic and its extensions can support answer-set programming in the same way stable logic programming and disjunctive logic programming do. To this end, we introduce a logic based on the logic of propositional schemata and on a version of the Closed World Assumption. We call it the *extended logic of propositional schemata with CWA* (PS^+, in symbols). An important feature of the logic PS^+ is that it supports explicit modeling of constraints on cardinalities of sets. In the paper, we characterize the class of problems that can be solved by finite PS^+ theories. We implement a programming system based on the logic PS^+ and design and implement a solver for processing theories in PS^+. We present encouraging performance results for our approach — we show it to be competitive with *smodels*, a state-of-the-art answer-set programming system based on stable logic programming.

1 Introduction

Logic is most commonly used in declarative programming and knowledge representation as follows. To solve a problem we represent its constraints and the relevant background knowledge as a theory in the language of some logic. We formulate the goal (the statement of the problem) as a formula of the logic. We then use proof techniques to decide whether this formula follows from the theory. A proof of the formula, variable substitutions or both determine a solution.

Recently, an alternative way in which logic can be used in computational knowledge representation has emerged from studies of nonstandard variants of logic programming such as logic programming with negation and disjunctive logic programming [MT99,Nie99]. This alternative approach is rooted in semantic notions and is based on methods to compute models. To represent a problem, we design a finite theory so that its *models* (and not proofs or variable substitutions) determine problem solutions (answers). To solve the problem, we compute models of the corresponding theory[1]. This model-based approach is now often referred to as *answer-set programming* (or ASP).

[1] We commonly restrict the language by disallowing function symbols to guarantee finiteness of models of finite theories. In the present paper, we also adopt this assumption.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 138–153, 2001.

Logic programming with stable model semantics [GL88] (*stable logic programming* or *SLP*, in short) is an example of an ASP formalism [MT99]. In SLP, we represent problem constraints by a fixed program (independent of problem instances). We represent a specific instance of the problem (input data) by a collection of ground atoms. To solve the problem, we find stable models of the program formed jointly by the two components. To this end, we first *ground* it (compute its equivalent propositional representation) and, then, compute stable models of this grounded propositional program. Thanks to the emergence of fast systems to compute stable models of propositional logic programs, such as *smodels* [NS00], SLP is quickly becoming a viable declarative programming environment for computational knowledge representation. Disjunctive logic programming with the semantics of answer sets [GL91] is another logic programming formalism that fits well into the answer-set programming paradigm. An effective solver for computing answer sets of disjunctive programs, *dlv*, is available [ELM$^+$98] and its performance is comparable with that of *smodels*.

Our goal in this paper is to propose answer-set programming formalisms based on propositional logic and its extensions. Our approach is motivated by recent improvements in the performance of satisfiability checkers. Researchers developed several new and fast implementations of the basic Davis-Putnam method such as *satz* [LA97] and *relsat* [BS97]. A renewed interest in local-search techniques resulted in highly effective (albeit incomplete) satisfiability checkers such as *WALKSAT* [SKC94], capable of handling large CNF theories, consisting of millions of clauses. Improvements in the performance resulted in an expanding range of applications of satisfiability checkers, with planning being one of the most spectacular examples [KMS96,KS99].

The way in which propositional satisfiability solvers are used in planning [KMS96] clearly fits the ASP paradigm. Planning problems are encoded as propositional theories so that models correspond to plans. In our paper, we extend ideas proposed in [KMS96] in the domain of planning and show that propositional satisfiability can be used as the foundation of a general purpose ASP system. To this end, we propose a logic to serve as a modeling language. This logic is a modification of the logic of propositional schemata [KMS96]; we explicitly separate theories into data and program, and use a version of Closed World Assumption (CWA) to define the semantics. This logic is nonmonotonic. We call it the *logic of propositional schemata with CWA* (or, PS^{cwa}).

The logic PS^{cwa} offers only basic logical connectives to help model problem constraints. We extend logic PS^{cwa} to support direct representation of constraints involving cardinalities. Examples of such constraints are: "at least k elements from the list must be in the model" or "exactly k elements from the list must be in the model". They appear commonly in statements of constraint satisfaction problems. We refer to this new logic as *extended logic of propositional schemata with Closed World Assumption* and denote it by PS^+.

In the paper we characterize the class of problems that can be solved by *finite* PS^+ theories. In other words, we determine the expressive power of the logic

PS^+. Specifically, we show that it is equal to the expressive power of function-free logic programming with the stable-model semantics.

For processing, theories in PS^+ could be compiled into propositional theories and "off-the-shelf" satisfiability checkers could be used for processing. However, propositional representations of constraints involving cardinalities are usually very large and the sizes of the compiled theories limit the effectiveness of satisfiability checkers, even the most advanced ones, as processing engines. Thus, we argue against the compilation of the cardinality constraints. Instead, we propose an alternative approach. We design a "target" propositional logic for the logic PS^+ (propositional logic PS^+). In this logic, cardinality constraints have explicit representations and, therefore, do not need to be compiled any further. We develop a satisfiability checker for the propositional logic PS^+ and use it as the processing back-end for the logic PS^+. Our solver is designed along the same lines as most satisfiability solvers implementing the Davis-Putnam algorithm but it takes a direct advantage of the cardinality constraints explicitly present in the language.

Experimental results on the performance of the overall system are highly encouraging. We obtain concise encodings of constraint problems and the performance of our solver is competitive with the performance of *smodels* and of state-of-the-art complete satisfiability checkers. Our work demonstrates that building propositional solvers capable of processing of high-level constraints is a promising research direction for the area of propositional satisfiability.

Our paper is organized as follows. In the next section we introduce the logic PS^{cwa} — a fragment of the logic PS^+ without cardinality constraints. We determine the expressive power of the logic PS^{cwa} in Section 3. We discuss the full logic PS^+ in Section 4. In the subsequent section we discuss implementation details and experimental results. The last section of the paper contains conclusions and comments on the future work.

2 Basic Logic PS^{cwa}

Our approach is based on the logic of *propositional schemata*. The syntax of this logic is that of first-order logic without function symbols. The semantics is that of *Herbrand interpretations* and *models*, which we identify with subsets of the *Herbrand base*. In the paper we consider only those theories in which at least one constant symbol appears. Among all formulas in the language, of main interest to us are *clauses*, that is, expressions of the form

$$a_1 \wedge \ldots \wedge a_m \Rightarrow B_1 \vee \ldots \vee B_n, \tag{1}$$

where each a_i is an atom and each B_j is an atom or an expression of the form $\exists Y b(s)$, where $b(s)$ is an atom and Y is a tuple of (not necessarily all) variables appearing in $b(s)$. Each of m and n (or both) may equal 0. If $m = 0$, we replace the conjunct in the antecedent of the clause with a special symbol **T** (*truth*). If $n = 0$, we replace the empty disjunct in the consequent of the clause with a special symbol **F** (*contradiction*). We assume that each clause is universally

quantified and drop the universal quantifiers from the notation. We further simplify the notation by replacing each expression $\exists Y b(s)$ in the antecedent by $b(s')$, where in s' we write a special symbol '_' for each variable from Y in s.

Let T be a *finite* theory consisting of clauses. For a formula $B = \exists Y b(s)$ appearing in the consequent of a clause in T, we define B^e to be the disjunction $B^e = \quad b(s^1) \vee \ldots \vee b(s^k)$, where s^i, $1 \leq i \leq k$, range over all term tuples that can be obtained from s by replacing variables in Y with constants appearing in T. Since T is finite, the disjunction is well defined (it has only finitely many disjuncts).

For a clause $C \in T$ of the form (1), we define a clause C^e by

$$C^e = \quad a_1 \wedge \ldots \wedge a_m \Rightarrow B_1^e \vee \ldots \vee B_n^e, \tag{2}$$

A *ground instance* of C is any formula obtained from C^e by replacing every variable in C^e by a constant appearing in T (different occurrences of the same variable must be replaced by the same constant). We define the *grounding* of T, $gr(T)$ as the collection of all ground instances of clauses in T, except for tautologies; they are not included in $gr(T)$. We have the following well-known result.

Proposition 1. *Let T be a finite clausal theory. Then a set of ground atoms M is a Herbrand model of T if and only if M is a (propositional) model of $gr(T)$.*

The language may contain several *predefined* predicates and function symbols such as the equality operator and arithmetic comparators and operations. We assign to these symbols their standard interpretation. However, we emphasize that the domains are restricted only to those constants that appear in a theory.

We evaluate all expressions involving predefined function symbols and all atoms involving predefined relation symbols in the grounding process. If any argument of a predefined relation is not of the appropriate type, we interpret the corresponding atom as false. If a function yields as a result a constant that does not appear in the theory or if one of its arguments is not of the required type, we also interpret the corresponding atom as false. We then eliminate tautologies and simplify the remaining clauses by removing true "predefined" atoms from the antecedents and false "predefined" atoms from the consequents.

Let us consider an example. Let T be a theory consisting of the following two clauses:

$$C_1 = \quad q(b, c) \Rightarrow p(a)$$
$$C_2 = \quad p(X) \Rightarrow (\exists Y q(X, Y)) \vee (X = a).$$

There are three constants, a, b and c, and two predicate symbols, p and q, in the language. Symbols X and Y denote variables. The clause C_2 can also be written (using the simplified notation) as

$$C_2 = \quad p(X) \Rightarrow q(X, \_) \vee (X = a).$$

To compute $gr(T)$ we need to compute all ground instances of C_2 (C_1 is itself its only ground instance). First, we compute the formula C_2^e:

$$C_2^e = \quad p(X) \Rightarrow q(X, a) \vee q(X, b) \vee q(X, c) \vee (X = a).$$

To obtain all ground instances of C_2 (or C_2^e), we replace X with a, b and c. The first substitution results in a tautology (due to occurrence of '$a = a$' in the consequent of the clause). Two other substitutions yield the following two ground instances of C (we drop atoms '$b = a$' and '$c = a$' from the consequents; they are false by the standard interpretation of equality):

$$p(b) \Rightarrow q(b,a) \vee q(b,b) \vee q(b,c)$$
$$p(c) \Rightarrow q(c,a) \vee q(c,b) \vee q(c,c).$$

These two clauses together with C_1 form $gr(T)$. The sets of ground atoms $\{p(a), q(b,c)\}$ and $\{p(b), p(c), q(b,a), q(c,c)\}$ are two examples of models of T (or $gr(T)$).

In order for the logic of propositional schemata to be useful as a programming tool, we modify it to separate input data from the program encoding the problem to be solved. We distinguish in the set of predicates Pr of the language a subset, Pr'. We call its elements *data* predicates. We assume that predefined predicates are *not* data predicates. All predicates other than data predicates and predefined predicates are called *program* predicates. A *theory* of our logic is a pair (D, P), where D is a *finite* collection of ground atoms whose predicate symbols are data predicates (*data*), and P is a *finite* collection of clauses (a *program*).

To define the semantics for the logic, we use grounding and a form of CWA. We say that a set of ground atoms (built of data and program predicates) is a *model* of a theory (D, P) if

M1: M is a model of $gr(D \cup P)$ (or, equivalently, M is an Herbrand model of $D \cup P$), and

M2: for every ground atom $p(t)$ such that $p \in Pr'$ (p is a data predicate), $p(t) \in M$ if and only if $p(t) \in D$.

We call the logic described above the *logic of propositional schemata with CWA* and denote it by PS^{cwa}. Due to (M2), not every model of $gr(D, P)$ is a model of (D, P). Consequently, one can show that our logic is nonmonotonic. This difference between the logic of propositional schemata and the logic PS^{cwa}, while seemingly small, has significant consequences for the expressive power of the logic and its applicability as a programming tool.

Before addressing these two issues, let us consider an example. Let A and B be two disjoint and finite sets. We define $D = \{p_1(a) \colon a \in A\} \cup \{p_2(b) \colon b \in B\}$. We define P to consist of two clauses:

Ex1: $q_1(X) \Rightarrow p_1(X)$ Ex2: $q_2(X) \Rightarrow p_2(X)$.

The constants are elements of $A \cup B$; X is a variable. The predicates are p_1, p_2, q_1 and q_2. The first two are data predicates.

By (M2), each model of a PS^{cwa} theory (D, P) contains D. However, it does not contain any ground atom $p_1(b)$, where $b \in B$, nor any ground atom $p_2(a)$, where $a \in A$. Each ground instance of the clause (Ex1) is of the form $q_1(c) \Rightarrow p_1(c)$, where c is a constant ($c \in A \cup B$). Since $p_1(c) \in M$ if and only if $c \in A$, it follows that if $q_1(c) \in M$, then $c \in A$. Similarly, we obtain that if $q_1(c) \in M$, then $c \in A$. Thus, M is a model of (D, P) if and only if $M = D \cup \{q_1(a) \colon a \in A'\} \cup \{q_2(b) \colon b \in B'\}$, for some $A' \subseteq A$ and $B' \subseteq B$.

Let us choose an element from A, say a_0, and an element from B, say b_0. Let us then add to P the clause

Ex3: $p_1(a_0) \Rightarrow p_1(b_0)$

We denote the new program by P'. The PS^{cwa} theory (D, P) has no models even though $gr(D, P)$ is propositionally consistent. The reason is that all propositional models satisfying $gr(D, P)$ contain $p_1(b_0)$. Thus, none of these models satisfies condition (M2). This example illustrates that our semantics is different from circumscription as circumscription preserves consistency. Circumscription applied to p_1 would result in models in which the extension of p_1 in D would be minimally extended by one more constant b_0. Our (strong) minimization principle does not allow for any additions to the extension of data predicates. Intuitively, it is exactly as it should be. **Data predicates are meant to represent input data. The program should not be able to extend it.**

Logic PS^{cwa} is a tool to model problems. To illustrate this use of the logic, we show how to encode the *vertex-cover* problem for graphs. Let $G = (V, E)$ be a graph. A set $W \subseteq V$ is a *vertex cover* of G if for every edge $\{x, y\} \in E$, x or y (or both) are in W. The vertex-cover problem is defined as follows: given a graph $G = (V, E)$ and an integer k, decide whether G has a vertex cover with no more than k vertices.

For the vertex-cover problem the input data is described by the following set of ground atoms:

$D_{vc} = \{vtx(v) \colon v \in V\} \cup \{edge(v, w) \colon \{v, w\} \in E\} \cup \{size(k)\} \cup \{pos(i) \colon 1, \ldots, n\}$.

This set specifies the set of vertices and the set of edges of an input graph. It provides the limit on the size of a vertex cover sought. Lastly, it uses a predicate *pos* to specify a range of *integers* that will be used to label vertices. The problem itself is described by the program P_{vc}:

VC1: $vpos(I, X) \Rightarrow vtx(X)$
VC2: $vpos(I, X) \Rightarrow pos(I)$
VC3: $vtx(X) \Rightarrow vpos(\_, X)$
VC4: $vpos(I, X) \wedge vpos(J, X) \Rightarrow I = J$
VC5: $vpos(I, X) \wedge vpos(I, Y) \Rightarrow X = Y$
VC6: $edge(X, Y) \wedge vpos(I, X) \wedge vpos(J, Y) \wedge size(K) \Rightarrow (I \leq K) \vee (J \leq K)$

(VC1) and (VC2) ensure that $vpos(i, x)$ is false if i is not an integer from the set $\{1, \ldots, n\}$ or if x is not a vertex. (VC3)-(VC5) together enforce that the atoms $vpos(i, x)$ that are true in a model of the PS^{cwa} theory (D_{vc}, P_{vc}) define a permutation of the vertices in V. Finally, (VC6) ensures that each edge has at least one vertex assigned by *vpos* to positions $1, \ldots, k$ (in other words, that vertices labeled $1, \ldots, k$ form a vertex cover). The correctness of this encoding is formally established in the following result.

Proposition 2. *Let $G = (V, E)$ be an undirected graph and let k be a positive integer. A set of vertices $\{w_1, \ldots, w_k\} \subseteq V$ is a vertex cover of G if and only if $M = D_{vc} \cup \{vpos(i, w_i) \colon i = 1, \ldots, k\}$ is a model of the theory (D_{vc}, P_{vc}).*

For another example, we will consider the n-queens problem, that is, the problem of placing n queens on a $n \times n$ chess board so that no queen attacks another.

In this case, the representation of input data describes the set of row and column indices:
$D_{nq} = \{pos(i)\colon 1, \ldots, n\}$.
The problem itself is described by the program P_{nq}. The predicate q describes a distribution of queens on the board: $q(x, y)$ is true precisely when there is a queen in the position (x, y).

nQ1: $q(R, C) \Rightarrow pos(R)$
nQ2: $q(R, C) \Rightarrow pos(C)$
nQ3: $q(R, C1) \wedge q(R, C2) \Rightarrow C1 = C2$
nQ4: $q(R1, C) \wedge q(R2, C) \Rightarrow R1 = R2$
nQ5: $q(R, C), q(R + I, C + I) \Rightarrow \mathbf{F}$
nQ6: $q(R, C), q(R + I, C - I) \Rightarrow \mathbf{F}$

The first two clauses ensure that if $q(r, c)$ is true in a model of (D_{nq}, P_{nq}) then r and c are integers from the set $\{1, \ldots, n\}$. The following two clauses enforce the constraint that no two queens are placed in the same row or the same column. Finally, the last two clauses guarantee that no two queens are placed on the same diagonal. As in the case of the vertex cover problem, also in this case we can formally show the correctness of this encoding.

These examples demonstrate that PS^{cwa} programs can serve as representations of computational problems. Two key questions arise: (1) what is the expressive power of the logic PS^{cwa}, and (2) how to use the logic PS^{cwa} as a practical computational tool. We address both questions in the remainder of the paper.

3 Expressive Power of PS^{cwa}

A *search* problem, Π, is given by a set of finite *instances*, D_Π, such that for each instance $I \in D_\Pi$, there is a finite set $S_\Pi(I)$ of all *solutions* to Π for the instance I [GJ79]. The graph-coloring, vertex-cover and n-queens problems considered in the previous section are search problems. More generally, all constraint satisfaction problems including basic AI problems such as planning, scheduling and product configuration can be cast as search problems.

We say that a PS^{cwa} program P *solves* a search problem Π if there exist:

1. A mapping d that can be computed in polynomial time and that encodes instances to Π as sets of ground atoms built of data predicates
2. A partial mapping *sol*, computable in polynomial time, that assigns to (some) sets of ground atoms solutions to Π (elements of $\bigcup_{I \in D_\Pi} S_\Pi(I)$)

such that for every instance $I \in D_\Pi$, $s \in S_\Pi(I)$ if and only if there exists a model M of the PS^{cwa} theory $(d(I), P)$ such that M is in the domain of the mapping *sol* and $sol(M) = s$.

A search problem Π is in the class *NP-search* if there is a nondeterministic Turing Machine TM such that (1) TM runs in polynomial time; (2) for every instance $I \in D_\Pi$, the set of strings left on the tape when accepting computations for I terminate is precisely the set of solutions $S_\Pi(I)$.

We now have the following theorem that determines the expressive power of the logic PS^{cwa}. Its proof is provided in the appendix.

Theorem 1. *A search problem Π can be solved by a PS^{cwa} program if and only if $\Pi \in$ NP-search.*

Decision problems can be viewed as special search problems. For the class of decision problems, Theorem 1 implies the following corollary (a counterpart to the result on the expressive power of DATALOG$^\neg$ [Sch95]).

Corollary 1. *A decision problem Π can be solved by a PS^{cwa} program if and only if Π is in NP.*

4 Extending PS^{cwa} — The Logic PS^+

We will now discuss ways to enhance effectiveness of logic PS^{cwa} as a modeling formalism and propose ways to improve computational performance. When considering the PS^{cwa} theories developed for the n-queens and vertex-cover problems one observes that these theories could be simplified if the language of the logic PS^{cwa} contained direct means to model constraints such as: "exactly one element is selected" or "at most k elements are selected".

With this motivation, we extend the language of the logic PS^{cwa}. We define a *c-atom* (cardinality atom) as an expression $m\{p(X, \_, Y)\}n$, where m and n are non-negative integers, X and Y are tuples of variables and p is a program predicate[2].

The interpretation of a c-atom is that for every ground tuples x and y that can be substituted for X and Y, at least m and at most n atoms from the set

$$\{p(x, c, y) \colon c \text{ is a constant appearing in the theory}\}$$

are true. One of m and n may be missing from the expression. If m is missing, there is no lower-bound constraint on the number of atoms that are true. If m is missing, there is no upper-bound constraint on the number of atoms that are true. It is also possible to have more "underscore" symbols in c-atoms. In such case, when forming the set of atoms on which cardinality constraints are imposed, all possible ways to replace the "underscore" symbols by constants are used.

An *extended clause* is a clause built of c-atoms. The notions of a *program* and *theory* are defined as in the case of the logic PS^{cwa}.

A theory in the extended syntax can be grounded, that is, represented as a set of propositional clauses, in a similar way as before. In particular, data and

[2] In our implementation, we support a somewhat more general form of c-atoms.

predefined predicates are treated in the same way and are subject to the same version of CWA that was used for the logic PS^{cwa}. While grounding, c-atoms are interpreted as explained earlier. Grounding allows us to lift the semantics of propositional logic to the theories in the extended syntax. We call the resulting logic the *extended logic* PS^{cwa} and denote it by PS^{+}.

In the logic PS^{+} we can encode the vertex cover problem in a more straightforward and more concise way. Namely, the problem can be represented without the need for integers to label the vertices of an input graph! This new representation (D'_{vc}, P'_{vc}) is given by:
$D'_{vc} = \{vtx(v)\colon v \in V\} \cup \{edge(v,w)\colon \{v,w\} \in E\}$,
and $P'_{vc} =$

VC′1: $invc(X) \Rightarrow vtx(X)$
VC′2: $\{invc(\_)\}k$
VC′3: $edge(X,Y) \Rightarrow invc(X) \vee invc(Y)$.

Atoms $invc(x)$ that are true in a model of the PS^{cwa} theory (D'_{vc}, P'_{vc}) define a set of vertices that is a candidate for a vertex cover. (VC′2) guarantees that no more than k vertices are included. (VC′3) enforces the vertex-cover constraint.

We close this section with an observation on the expressive power of the logic PS^{+}. Since it is a generalization of the logic PS^{cwa}, it can capture all problems that are in the class NP-search. On the other hand, the problem of computing models of a PS^{+} theory with a fixed program part is an NP-search problem, it follows that the expressive power of the logics PS^{+} does not extend beyond the class NP-search. In other words, the logic PS^{+} also captures the class NP-search.

5 Computing with PS^{+} Theories

To process PS^{+} theories, one approach is to ground them into collections of propositional clauses. However, CNF representations of c-atoms may be quite large; the constraint "at most n atoms in the set $\{p_1, \ldots, p_k\}$ are true", is captured by $\theta(k^{n+1})$ clauses $p_{i_1}, \ldots, p_{i_{n+1}} \Rightarrow \mathbf{F}$, one for each $(n+1)$-element subset $\{p_{i_1}, \ldots, p_{i_{n+1}}\}$ of $\{p_1, \ldots, p_k\}$.

Thus, we propose another approach. The idea is to develop an extension of propositional logic representing c-atoms directly. Let At be a set of propositional variables. By a *propositional c-atom* we mean any expression of the form $m\{p_1, \ldots, p_k\}n$, where m and n are non-negative integers and $p_1, \ldots, p_k$ are atoms in At (one of m and n may be missing). By an *extended propositional clause* we mean an expression of the form

$$C = \quad A_1 \wedge \ldots \wedge A_s \Rightarrow B_1 \vee \ldots \vee B_t,$$

where all A_i and B_i are propositional c-atoms.

Let $M \subseteq At$ be a set of atoms. We say that M *satisfies* a generalized atom $m\{p_1, \ldots, p_k\}n$ if

$$m \leq |M \cap \{p_1, \ldots, p_k\}| \leq n.$$

Further, M *satisfies* a generalized clause C if M satisfies at least one atom B_j or does not satisfy at least one atom A_i. We call the resulting logic the *propositional logic* PS^+. Clearly, M satisfies an atom $1\{p\}1$ if and only if $p \in M$. Thus, the propositional logic PS^+ extends the (clausal) propositional logic.

Theories of the logic PS^+ can be grounded in the extended propositional logic by generalizing the approach described in Section 2. We represent c-atoms as propositional c-atoms and avoid a blow-up in the size of the representation. The problem is that SAT checkers cannot now be used to resolve the satisfiability of the extended propositional logic as they are not designed to work with the extended syntax.

It is clear, however, that the techniques developed in the area of SAT checkers can be extended to the propositional logic PS^+. We have developed a Davis-Putnam like procedure, *aspps*, that finds models of propositional PS^+. We also developed a program *psgrnd* that accepts theories in the syntax of the logic PS^+ and grounds them into propositional PS^+ theories. Thus, the two programs together can be used as a processing mechanism for an answer-set programming system based on the logic PS^+. The programs *psgrnd* and *aspps* are available at `http://www.cs.uky.edu/ai/aspps/`.

In our experiments we considered the vertex-cover problem and several combinatorial problems including n-queens problem, pigeonhole problem and the problem to compute Schur numbers. All our experiments were performed on a Pentium III 500MHz machine running linux.

We were mostly interested in comparing the performance of our system *psgrnd/aspps* with that of *smodels*. The reason is that both programs accept similar syntax and allow for very similar modeling of constraints. We also experimented with a satisfiability checker *satz*.

In the case of vertex cover, for each $n = 50$, 60, 70 and 80, we randomly generated 100 graphs with n vertices and $2n$ edges. For each graph G, we computed the minimum size k_G for which the vertex cover can be found. We then tested *aspps*, *smodels* and *satz* on all the instances (G, k_G). The results represent the average execution times Encodings we used for testing *aspps* and *smodels* where based on the clauses (VC′1) - (VC′3). For *satz* we used encodings based on the clauses (VC1) - (VC6) (cardinality constraints cannot be handled by *satz*).

A propositional CNF theory obtained by grounding the program (VC1) - (VC6), has $\Theta(n^2)$ atoms, $\Theta(mn^2)$ clauses and its total size is also $\Theta(mn^2)$. For input instances we used in our experiments, these theories were of such large sizes (over one million rules in the case of graphs with 80 vertices) that *satz* did not terminate in the time we allocated (5 minutes). Thus, no times for *satz* are reported. On the other hand, since the propositional PS^+ theory obtained by grounding the PS^+ program (VC′1) - (VC′3) has only $\Theta(m + n)$ clauses (a few hundred clauses for graphs with 80 vertices) and its total size has the same asymptotic estimate. This is dramatically less than in the case of theories *satz* had to process. Both *aspps* and *smodels* performed very well, with *aspps* being about three times faster than *smodels*. The timing results are summarized in Table 1.

Table 1. Timing results (in seconds) for the vertex-cover problem.

n	50	60	70	80
aspps	0.04	0.22	1.26	6.45
smodels	0.12	0.76	4.14	22.35

For the n-queens problem, our solver performed exceptionally well. It scaled up much better than *smodels* both in the case when we were looking for one solution and when we wanted to compute all solutions. In particular, our program found a solution to the 36 queens problem in 0.97 sec. It also outperformed *satz*.

Table 2. Timing results (in seconds) for the n-queen problem.

# of queens	18	19	20	21	22	23
aspps	0.02	0.02	0.07	0.07	0.11	0.12
smodels	2.35	1.28	13.25	19.31	167.1	380.35
satz	1.16	0.61	4.35	0.95	28.64	1.42

The pigeonhole problem consists of showing that it is not possible to place p pigeons in h holes if $p > h$. For this problem *aspps* showed the best performance — about three times faster than the other two solvers (all programs showed a similar rate of growth in the execution time).

Table 3. Timing results (in seconds) for the pigeonhole problem

(p, h)	(9,8)	(10,9)	(11,10)	(12,11)
aspps	0.59	5.63	60.08	702.02
smodels	2.7	21.56	219.99	2469.97
satz	1.87	17.28	178.20	2044.42

The Schur problem consists of placing n numbers $1, 2, \ldots, n$ in k bins so that the set of numbers assigned to a bin is not closed under sums. That is, for all numbers x, y, z, $1 \leq x, y, z \leq n$, if x and y are in a bin b, then z is not in b (x and y need not be distinct). The Schur number $S(k)$ is the maximum number n for which such a placement is still possible.

We considered the problem of the existence of the placement for $k = 4$ and values of n ranging from 40 to 45. For $n \leq 44$ all programs found a "Schur" placement. However, no "Schur" placement exists for $n = 45$ (and higher values of n). All programs were able to establish the non-existence of solutions for $n = 45$ (but the times grew significantly). Our results summarizing the performance of our system and *smodels* on the theories encoding the constraints of the problem are shown in Table 4. *aspps* and *satz* seem to performed better than *smodels*, with *satz* being slightly faster for values of n closer to the Schur number.

In the case of the last three problems, it was possible to eliminate cardinality constraints without significant increase in the size of grounded theories. As a result, *satz* performed well.

Table 4. Timing results (in seconds) for the Schur-number problem.

n	40	41	42	43	44	45
aspps	0.03	0.03	0.03	0.03	1.83	54.5
smodels	0.3	0.38	0.32	0.36	35.8	>1500
satz	0.21	0.23	0.24	0.25	0.96	20.4

6 Conclusions

Our work demonstrates that propositional logic and its extensions can support answer-set programming systems in a way in which stable logic programming and disjunctive logic programming do[3]. In the paper we described logic PS^+ that can be used to this end. We presented an effective implementation of a grounder, *psgrnd*, and a solver, *aspps*, for processing theories in the logic PS^+. Our experimental results are encouraging. Our system is competitive with *smodels*, and in many cases outperforms it. It is also competitive with satisfiability solvers such as *satz*.

The results of the paper show that programming front-ends for constraint satisfaction problems that support explicit coding of complex constraints facilitate modeling and result in concise representations. They also show that solvers such as *aspps* that take advantage of those concise encodings and process high-level constraints directly, without compiling them to simpler representations, exhibit very good computational performance. These two aspects are important. Satisfiability checkers often cannot effectively solve problems simply due to the fact that encodings they have to work with are large. For instance, for the vertex-cover problem for graphs with 80 vertices and 160 edges, *aspps* has to deal with theories that consist of a few hundred of rules only. In the same time pure propositional encodings of the same problem contain over one million clauses — a factor that undoubtedly is behind much poorer performance of *satz* on this problem.

Our work raises new questions. Further extensions of logic PS^+ are possible. For instance, constraints that impose other conditions on set cardinalities than those considered here (such as, the *parity* constraint) might be included. We will pursue this direction. Similarly, there is much room for improvement in the area of solvers for the propositional logic PS^+. In particular, we will study local search algorithms as possible satisfiability solvers for propositional PS^+ theories.

Finally, we note that the experimental results presented here are meant to show that *aspps* is competitive with other solvers and, we think, they demonstrate this. However, these results are still too fragmentary to provide basis for any conclusive comparison between the three solvers tested. Such a comparison is further complicated by the fact that the same problem may have several different

[3] We point out, though, that stable logic programming and disjunctive logic programming directly support negation-as-failure and, consequently, yield more direct solutions to some knowledge representation problems such as, for example, the frame problem.

encodings with different computational properties. Developing the methodology for comparing solvers designed to work with different formal systems is a challenging problem for builders of constraint solvers and declarative programming systems.

Acknowledgments. This work was partially supported by the NSF grants CDA-9502645, IRI-9619233 and EPS-9874764.

References

[Apt90] K. Apt. Logic programming. In J. van Leeuven, editor, *Handbook of theoretical computer science*, pages 493–574. Elsevier, Amsterdam, 1990.

[BS97] R.J. Bayardo, Jr and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*. MIT Press, 1997.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and data bases*, pages 293–322. Plenum Press, New York-London, 1978.

[ELM+98] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A KR system `dlv`: Progress report, comparisons and benchmarks. In *Proceeding of the Sixth International Conference on Knowledge Representation and Reasoning (KR '98)*, pages 406–417. Morgan Kaufmann, 1998.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness*. W.H. Freeman and Co., San Francisco, Calif., 1979.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[KMS96] H.A. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proceedings of KR-96*, pages 374–384. Morgan Kaufmann, 1996.

[KS99] H.A. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *Proceedings of IJCAI-99*, San Mateo, CA, 1999. Morgan Kaufmann.

[LA97] C.M. Li and M. Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, 1997.

[Llo84] J. W. Lloyd. *Foundations of logic programming*. Symbolic Computation. Artificial Intelligence. Springer-Verlag, Berlin-New York, 1984.

[MR01] W. Marek and J.B. Remmel. On the foundations of answer-set programming. In *Answer-Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*. AAAI Press, 2001. Papers from the 2001 AAAI Spring Symposium, Technical Report SS-01-01.

[MT99] V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[Nie99] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[NS00] I. Niemelä and P. Simons. Extending the smodels system with cardinality and weight constraints. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.

[Sch95] J. Schlipf. The expressive powers of the logic programming semantics. *Journal of the Computer Systems and Science*, 51(1):64–86, 1995.

[SKC94] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, USA, 1994.

Appendix

We will present here a sketch of a proof of our main result concerning the expressive power of the logic PS^{cwa}. The proof relies on some basic notions from logic programming (we refer the reader to [Apt90,Llo84] for details).

We restrict our discussion to function-free languages (the case relevant to our logic PS^{cwa}). Given a predicate language $\mathcal{L}$ (as defined in Section 2), a *logic program clause* over this language is an expression r of the form

$$r = \quad p(t) \leftarrow q_1(t_1), \ldots, q_m(t_m), \mathbf{not}(q_{m+1}(t_{m+1})), \ldots, \mathbf{not}(q_{m+n}(t_{m+n}))$$

where $p, q_1, \ldots, q_{m+n} \in Pr$, (we assume that p is not a predefined predicate), and $t, t_1, \ldots, t_{m+n}$ are term tuples with the arity matching the arity of the corresponding predicate symbol. We call the atom $p(t)$ the *head* of the rule r and denote it by $h(r)$. For a rule r we also define

$$B(r) = q_1(t_1) \wedge \ldots q_m(t_m) \wedge \neg q_{m+1}(t_{m+1}) \wedge \ldots \wedge \neg q_n(t_n)$$

We will be interested in supported models of logic programs. Without loss of generality, we will restrict our attention to programs in the *normal form*. That is, we assume that (1) the head of each rule is of the form $p(t)$, where t is a tuple of variables, and (2) if p appears in the head of two rules, the heads of these two rules are exactly the same (the same tuple of variables appear in both of them) [Cla78,Apt90].

Let P be a program in the normal form. For each predicate symbol $p \in Pr(P)$, we define a formula $cc(p)$ by:

$$cc(p) = \quad p(X) \Leftrightarrow \bigvee \{\exists Y_r B(r) \colon r \in P', h(r) = p(X)\},$$

where X is a tuple of variables and Y_r is the tuple of variables occurring in the body of r but not in the head of r (we exploit the normal form of P here). We define the *completion* of P, $CC(P)$, by setting $CC(P) = \{cc(p) \colon p \in Pr\}$.

The Clark's completion is important as it allows us to characterize *supported* models of a logic program [Apt90]. Namely, we have the following result.

Theorem 2. *Let P be a logic program. A set of ground atoms M is a supported model of P if and only if it is a Herbrand model of $CC(P)$.*

We now have the following theorem.

Theorem 3. *Let P be a logic program in the normal form. Let Pr be the set of predicates appearing in P and let Pr' be the set of predicates of P that do not appear in the heads of rules in P. There is a PS^{cwa} theory $T(P)$ such that for every set of ground atoms D over predicates from Pr', a set of ground atoms M is a supported model of $D \cup P$ if and only if $M = M' \cap HB(P)$ for some model M' of the PS^{cwa} theory $(D, T(P))$.*

Proof: (Sketch) To define $T(P)$, we consider the completion $CC(P)$ of P. The idea is to take for $T(P)$ an equivalent clausal representation of $CC(P)$.

We build such representation as follows. Let p be a predicate symbol in $Pr \setminus Pr'$. The completion $CC(P)$ contains the formula

$$cc(p) = \quad p(X) \Leftrightarrow \bigvee \{\exists Y_r B(r) \colon r \in P, h(r) = p(X)\},$$

where X is a tuple of variables and Y_r is the tuple of variables occurring in the body of r but not in the head of r. For each rule $r \in P$ such that p occurs in $h(r)$, we introduce a new predicate symbol d_r, of the same arity $|X| + |Y_r|$. We define a theory $T'(P)$ to consist of the following formulas (we recall that $B(r)$ stands for the conjunction of the literals from the body of r):

$$\psi(r) = \quad d_r(X, Y_r) \Leftrightarrow B(r),$$

where $p \in Pr \setminus Pr'$, $r \in P$ and p occurs in the head of r, and

$$cc'(p) = \quad p(X) \Leftrightarrow \bigvee \{\exists Y_r d_r(X, Y_r) \colon r \in P, h(r) = p(X)\},$$

where $p \in Pr \setminus Pr'$.

It is clear that the theory $T'(P)$ is equivalent to $CC(P)$ (modulo new ground atoms). That is, $M \subseteq HB(P)$ is a model of $CC(P)$ if and only if $M = M' \cap HB(P)$, for some model M' of $T'(P)$.

One can show that $T'(P)$ can be rewritten (in polynomial time) into an equivalent clausal form, $T(P)$. Consequently, $T(P)$ is equivalent to $CC(P)$ (modulo ground atoms $d_r(t)$). It is now a routine task to verify that the theory $T(P)$ satisfies all the requirements of the statement of the theorem. □

Using the terminology introduced here we will now prove Theorem 1 from Section 3.

Theorem 4. *A search problem Π can be solved by a finite PS^{cwa} program if and only if $\Pi \in$ NP-search.*

Proof: (Sketch) In [MR01] it is proved that every NP-search problem can be solved uniformly by a finite logic program under the supported-model semantics. Since the theory $T(P)$ can be constructed in polynomial time, it follows

by Theorem 3 that every search problem in NP-search can be solved by a finite PS^{cwa} program. Conversely, for every fixed program P, the problem of computing models of a PS^{cwa} theory (D, P) (D is the input) is clearly in the class NP-search. Thus, only search problem in the class NP-search can be solved by finite PS^{cwa} programs. Hence, the assertion follows. □

Prediction of Regular Search Tree Growth by Spectral Analysis

Stefan Edelkamp

Institut für Informatik
Georges-Köhler-Allee, Gebäude 51
79110 Freiburg
edelkamp@informatik.uni-freiburg.de

Abstract. The time complexity analysis of the IDA* algorithm has shown that predicting the growth of the search tree essentially relies on only two criteria: The number of nodes in the brute-force search tree for a given depth and the equilibrium distribution of the heuristic estimate. Since the latter can be approximated by random sampling, we accurately predict the number of nodes in the brute-force search tree for large depth in closed form by analyzing the spectrum of the problem graph or one of its factorization.
We further derive that the asymptotic brute-force branching factor is in fact the spectral radius of the problem graph and exemplify our considerations in the domain of the $(n^2 - 1)$-Puzzle.

1 Introduction

Heuristic search is essential to AI, since it allows very large problem spaces to be traversed with a considerably small number of node expansions. Nevertheless, storing this number of nodes in memory, as required in the A* algorithm [5], often exceeds the resources available. This is bypassed in an iterative deepening version of A*, IDA* for short, that searches the tree expansion of the original state graph instead of the graph itself. IDA* [10] applies bounded depth-first traversals with an increasing threshold on A*'s node evaluation function. The tree expansion may contain several duplicate nodes such that low memory consumption is counterbalanced with a considerably high overhead in time.

Fortunately, due to simple search tree pruning rules and expressive heuristic estimates to direct the search process, duplicates in regular search spaces are rare such that IDA* has been very successfully applied to solve solitaire games like the $(n^2 - 1)$ Puzzle [9,12,15] and Rubik's Cube [11].

Korf, Reid and Edelkamp [14] have analyzed the IDA* algorithm to predict the search performance of IDA* in the number of node expansions for a specific problem. The main result is that assuming consistency[1] of the integral heuristic

[1] Consistent heuristic estimates satisfy $h(v) - h(u) + 1 \geq 0$ for each edge (u, v) in the underlying problem graph. They yield monotone node evaluations $f(u) = g(u) + h(u)$ on generating paths with length $g(u)$. Admissible heuristics are lower bound estimates that underestimate the goal distance for each state. Consistent estimates are admissible.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 154–168, 2001.

estimate in the limit of large c, the expected total number of node expansions with cost threshold c in one iteration of IDA* is equal to

$$\sum_{d=0}^{c} n^{(d)} P(c-d),$$

where $n^{(d)}$ is the number of nodes in the brute-force search tree with depth d and P is the equilibrium distribution defined as the probability distribution of heuristic values in the limit of large depth. More precisely, $P(h)$ is the probability that a randomly and uniformly chosen node of a given depth has a heuristic value less than or equal to h. In practice the equilibrium distribution for admissible heuristic functions will be approximated by random sampling [13]; a representative sample of the problem space is drawn and classified according to the integral heuristic evaluation function. The value $n^{(d)}$ for large depths d without necessarily exploring the search tree, can be approximated with the asymptotic brute-force branching factor; the number of nodes at one depth divided by the number of nodes in the next shallower depth, in the limit as the depth goes to infinity. The asymptotic heuristic branching factor is defined analogously on search tree levels for two occurring values on the node evaluation function f. In some domains we observe anomalies in the limiting behavior of the asymptotic branching factors, e.g., in the (n^2-1)-Puzzle and odd values of n it alternates between two different values [2].

The observation that a consistent heuristic estimate h affects the relative depth to a goal instead of the branching itself is supported by the fact that IDA*'s exploration is equivalent to undirected iterative deepening exploration in a re-weighted problem graph with costs $1 + h(v) - h(u)$ for all edges (u, v). The new node evaluation $f'(u_j)$ of node u_j on path $p = (s = u_1, \ldots, u_t = t)$ equals $\sum_{i=1}^{j-1} (1 + h(u_{i+1}) - h(u_i))$ and telescopes to the old merit $f(u_j)$ minus $h(s)$. Therefore, the heuristic is best understood as a bonus to the search depth. Moreover, since we have only altered edge weights, it is not surprising that for bounded heuristic estimates and large depth the asymptotic heuristic branching factor equals the asymptotic brute-force branching factor.

Our main result in this paper is that in undirected problem graphs the value of the number of nodes in depth d of the brute-force search can be computed effectively by analyzing the spectrum of the adjacency matrix for the problem graph. The analysis requires some results of linear algebra and an algorithm of applied mathematics. Since the problem graph is considered to be large for regular search spaces we show how to factorize the problem graph through an equivalence relation of same branching behavior. We take the (n^2-1)-Puzzle as the running example, discuss the generality of the results from various points of view: other problem domains, general, especially undirected graph structures, and predecessor pruning. Finally, we give concluding remarks and shed light on future research options.

2 Linear Algebra Basics

Linear Mappings and Bases. A mapping $f : V \to W$, with V, W being vector spaces over the field K (e.g. the set of real or the set of complex numbers) is *linear*, if $f(\lambda v + \mu w) = \lambda f(v) + \mu f(w)$ for all $v, w \in V$ and all $\lambda, \mu \in K$. A *basis* of a vector space V is a linear independent set of vectors that spans V. If the basis is finite, its cardinality defines the dimension $dim(V)$ of the vector space V, otherwise the dimension is said to be infinite.

Matrices and Basis-Transformations. Linear mappings of vector spaces of finite dimension can be represented as matrices, since there is an isomorphism that maps the set of all $(m \times n)$ matrices to the set of all linear mappings from V to W according to their respectively fixed bases, where $dim(V) = n$ and $dim(W) = m$. Usually, V equals W and in this case the linear mapping f is called *endomorphism*. A *basis-transformation* from basis $\mathcal{A}$ to $\mathcal{B}$ in the vector space V can be represented by a transformation matrix $C_{\mathcal{AB}}$ which is the inverse of $C_{\mathcal{BA}}$. Very often, $\mathcal{A}$ is the canonical basis. Computing the inverse C^{-1} of a matrix C can be achieved by elementary row transformations, that convert the $(n \times 2n)$ matrix $[C \mid I]$ into $[I \mid C^{-1}]$, with I being the identity matrix.

Similarity and Normal Forms. Two matrices A and B are *similar*, if there is a matrix C with $B = CAC^{-1}$. This is equivalent to the fact that there is an endomorphism f of V and two bases $\mathcal{A}$ and $\mathcal{B}$ with matrix A representing f according to $\mathcal{A}$ and B representing f according to $\mathcal{B}$. Similarity is an equivalence relation and one main problem in linear algebra is to derive a concise representative in the equivalence class of similar matrices, the *normal form*. A very simple form is the diagonal shape with non-zero values $\lambda_1, \ldots, \lambda_n$ only on the main diagonal. In this case, a matrix B is called *diagonizable* and can be written as $B = C \cdot \mathrm{diag}(\lambda_1, \ldots, \lambda_n) \cdot C^{-1}$. Unfortunately, not all matrices are diagonizable, especially when the linear mapping is defined on the set of real numbers. Even if the vector space defining field is the set of complex numbers, only *tridiagonizability* can be granted, in which matrix A may have non-zero components above the main diagonal. Further simplifications lead to the so-called *Jordan normal form*.

Eigenvalues and Eigenspaces. An endomorphism f of a vector space V over the field K contains an *eigenvalue* $\lambda \in K$, if there is a non-trivial vector $v \in V$, with $f(v) = \lambda v$. Any such non-trivial vector $v \in V$ with $f(v) = \lambda v$ is called *eigenvector*. If there is a basis $\mathcal{B}$ of eigenvectors, then the matrix representation according to $\mathcal{B}$ has a diagonal shape. In this case f is also called *diagonizable*. It can be shown that the eigenvalues are roots of the *characteristic equation* $P_f(\lambda) = \det(A - \lambda I) = 0$, where the determinant $\det(\mathrm{A})$ is defined as $\sum_{\sigma \in S_n} \prod_{i<j} (\sigma(j) - \sigma(i))/(j-i)) \cdot a_{1\sigma(1)} \cdot \ldots \cdot a_{n\sigma(n)}$ with S_n being the set of all n-permutations. If the polynomial $P_f(\lambda)$ factorizes, i.e. $P_f(\lambda) = const \cdot \prod_{i=1}^{k} (\lambda - \lambda_i)$, which is the case for matrices of complex numbers, the corresponding eigenspaces $E_f(\lambda_i)$ have to be computed. If then the number of occurring linear terms $(\lambda - \lambda_i)$

in $P_f(\lambda)$, the *algebraic multiplicity* of λ_i, equals the dimension of $E_f(\lambda_i)$, the *geometric multiplicity* of λ_i, then A is indeed *diagonizable*.

3 Partitioning the Search Space

The (n^2-1)-Puzzle is a sliding tile toy problem. It consists of (n^2-1) numbered tiles that can be slid into a single empty position, called the blank. The goal is to rearrange the tiles such that a certain goal position is reached. Figure 1 depicts possible end configurations of well-known instances to the (n^2-1)-Puzzle: For $n = 3$ we get the Eight-Puzzle, for $n = 4$ the Fifteen-Puzzle and for $n = 5$, the Twenty-Four-Puzzle is met. The state spaces for these problems grow

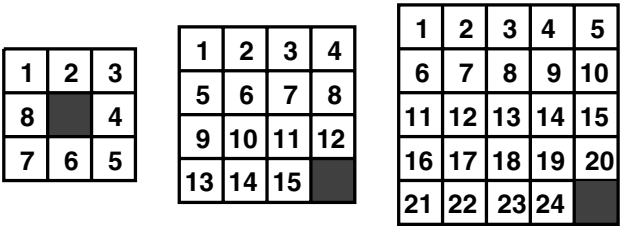


Fig. 1. The Eight-, Fifteen- and Twenty-Four-Puzzles.

exponentially. The exact number of reachable states (independent of the initial one) is $(n^2)!/2$ which resolves to approximately 10^5 states for the Eight-Puzzle, 10^{13} states in the Fifteen-Puzzle and 10^{25} states in the Twenty-Four-Puzzle.

We partition the search space S in classes $S_1, \ldots, S_k$, collecting states into groups with same branching behavior. In other words we devise an equivalence relation that partitions the state space into equivalence classes: two states are equivalent if their long term branching behavior coincides. All states in one equivalence class S_i, $i \in \{1, \ldots, k\}$, necessarily have the same node branching factor, defined as the number of children a node has in the brute-force search tree.

For the example of the (n^2-1)-Puzzle a partition is given by the following relation: two states are equivalent if the blank is at the same absolute position. Obviously the subtrees of such nodes are isomorphic, since the branching behavior of equivalent states has to be the same. A further reduction of the search tree is established by partitioning the search space with respect to symmetry. For the (n^2-1) Puzzle we establish three branching types: corner or c-nodes with node branching factor 2, side or s-nodes with node branching factor 3, and middle or m-nodes with node branching factor 4. However, does the long time node branching behavior depend on these node types only? In the Eight- and Fifteen-Puzzles this is the case, since for symmetry reasons all c, s and m nodes generate the same subtree structure. For the Twenty-Four-Puzzle, however, the

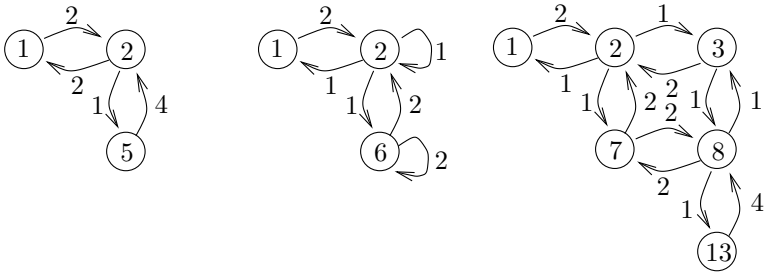


Fig. 2. Equivalence Graph for the Eight-, Fifteen- and Twenty-Four-Puzzles.

search tree of two side or two middle states may differ. For this case we need six classes with a blank at position 1,2,3,7,8, and 13 according to the tile labeling in Figure 2. In the general case the number of different node branching classes in the $(n^2 - 1)$ Puzzle is

$$\sum_{i=0}^{\lceil n/2 \rceil} i = \binom{\lceil n/2 \rceil}{2} = \lceil n/2 \rceil (\lceil n/2 \rceil - 1)/2.$$

This still compares well to a partition according to the n^2 equivalent classes in the first factorization (savings of a factor of about eight) and of course to the $(n^2)!/2$ states in the overall search space (exponential savings).

4 Equivalence Graph Structure

Utilizing this partition technique we define the weighted *equivalence graph* $\overline{G} = (\overline{V}, \overline{E}, w)$ as follows. The set of nodes $\overline{V}$ equals the set of equivalence classes and an edge e from class $S_i \in \overline{V}$ to $S_j \in \overline{V}$ with weight $w(e)$ is drawn, if every state in S_i leads to w states in class S_j. Obviously, the sum of all outgoing edges equals the node branching factor. Let $A_{\overline{G}}$ be the adjacency matrix with respect to the *equivalence graph* $\overline{G}$. Since the explorations in G and $\overline{G}$ span the same search-tree structure the search tree growth will be the same.

A generator matrix P for the population of nodes according to the given equivalence relation is defined by $P = A_{\overline{G}}^T$. More precisely, $P_{j,i} = l$ if a node of type i in a given level leads to l nodes of type j in the next level. We immediately infer that $N^{(d)} = PN^{(d-1)}$, with $N^{(d)}$ being the vector of nodes in depth d of the search tree. If $|| \cdot ||_1$ denotes the vector norm $||x||_1 = |x_1| + \ldots + |x_k|$ then the number of nodes $n^{(d)}$ in depth d is equal to $||N^{(d)}||_1$.

The asymptotic branching factor b (if it exists) is defined as the limit of $n^{(d)}/n^{(d-1)}$ for increasing d and equals the weighted product of the node frequencies $b = \sum_{i=1}^{k} b_i f_i$, where f_i is the fraction of nodes of class i with respect to the total number of nodes. As we will see, we can compute the branching factor analytically without actually determining node frequency values.

The first observation is that in case of convergence the asymptotic branching factor is not only met in the the overall search tree expansion but in every equivalence class. Since all frequencies of nodes converge we have that $b = \lim_{d\to\infty} N_i^{(d)}/N_i^{(d-1)}$, with $N_i^{(d)}$ being the number of nodes of class i in depth d, $i \in \{1, \ldots, k\}$. In other words, if the ratio of the cardinality of one equivalence class and the overall search space size settles and the search space size grows with factor b, then the equivalence class size itself grows with factor b.

We represent the fractions f_i as a distribution vector F. We first assume that this vector converges in the limit of large depth. The considerations for an analytical solution to the branching factor problem result in the equations $bF = FP$, where b is the asymptotic branching factor. In addition, we have the equation that the total of all node frequencies is one. The underlying mathematical issue is an eigenvalue problem. Transforming $bF = PF$ leads to $0 = (P - bI)F$ for the identity matrix I. The solutions for b are the roots of the characteristic equation $\det(P - bI) = 0$ where det is the determinant of the matrix. Since $\det(P - bI) = \det(P^T - bI)$ the transposition of the equivalence graph matrix $A_{\overline{G}}$ preserves the value of b. In case of the Eight-Puzzle $\det(P - bI)$ equals

$$\det \begin{pmatrix} 0-b & 2 & 0 \\ 2 & 0-b & 1 \\ 0 & 4 & 0-b \end{pmatrix} = 0.$$

This equation is equivalent to $b(4 - b^2) + 4b = 0$, yielding the following three solutions $-\sqrt{8} = -2.828427124$, 0, $\sqrt{8} = 2.828427124$. Experimental results show that the branching factor alternates every two depth values between 3 and $8/3 = 2.666666666$. Since $\sqrt{8}$ is the geometric mean of 3 and $8/3$ the value $\sqrt{8}$ is the proper choice for the asymptotic branching factor b of the brute-force search tree.

For the case of the Fifteen-Puzzle we have to calculate

$$\det \begin{pmatrix} 0-b & 2 & 0 \\ 1 & 1-b & 1 \\ 0 & 2 & 2-b \end{pmatrix} = 0,$$

which simplifies to $(1-b)(b-2)b+4b-4 = 0$. The solution to this equation are 1, $1 + \sqrt{5} = 3.236067978$, and $1 - \sqrt{5} = -1.236067978$. The value $1 + \sqrt{5}$ matches experimental data for the asymptotic branching factor.

For the Twenty-Four-Puzzle we have to solve

$$\det \begin{pmatrix} 0-b & 2 & 0 & 0 & 0 & 0 \\ 1 & 0-b & 1 & 1 & 0 & 0 \\ 0 & 2 & 0-b & 0 & 1 & 0 \\ 0 & 2 & 0 & 0-b & 2 & 0 \\ 0 & 0 & 1 & 2 & 0-b & 1 \\ 0 & 0 & 0 & 0 & 4 & 0-b \end{pmatrix} = 0.$$

The six eigenvalues are 0, 0, $\sqrt{3} = 1.732050808$, $-\sqrt{3} = -1.732050808$, $\sqrt{12} = 3.464101616$, and $-\sqrt{12} = -3.464101616$. Experiments show that for large depth the branching factor oscillates and that the geometric mean is 3.464101616.

We conclude that the asymptotic branching factor in the example problems is the *largest* eigenvalue of the adjacency matrix for the equivalence graph and that we observe anomalies if the largest eigenvalue has a negative counterpart of the same absolute value. In the following we will analyze the structure of the eigenvalue problem to show why this is the case.

5 Exact Prediction of Search Tree Size

The equation $N^{(d)} = PN^{(d-1)}$ can be unrolled to $N^{(d)} = P^d N^{(0)}$. We briefly sketch how to compute P^d for large d. We have seen that P is *diagonizable*, if there exists a invertible matrix C and a diagonal matrix Q with $P = CQC^{-1}$. This simplifies the calculation of P^d, since we have $P^d = CQ^dC^{-1}$ (the remaining terms $C^{-1}C$ cancel). By the diagonal shape of Q the value of Q^d is obtained by taking the matrix elements $q_{i,i}$ to the power of d. These elements are the eigenvalues of P. This connection is not surprising, since in case of convergence of the vector of node frequencies F we have seen that the branching factor itself is a solution to the eigenvalue problem $PF = bF$. We conclude that in case of *diagonizability* we can exactly predict the number of nodes of depth d by determining the set of eigenvalues of P.

In the example of the Eight-Puzzle the eigenvectors for the eigenvalues $-\sqrt{8}$, 0, and $\sqrt{8}$ are $(2, -\sqrt{8}, 1)^T$, $(-2, 0, 1)^T$, and $(2, \sqrt{8}, 1)^T$, respectively. Therefore, the basis-transformation matrix C is given by

$$C = \begin{pmatrix} 2 & -2 & 2 \\ -\sqrt{8} & 0 & \sqrt{8} \\ 1 & 1 & 1 \end{pmatrix}$$

with the following inverse

$$C^{-1} = 1/16 \begin{pmatrix} 2 & -\sqrt{8} & 4 \\ -2 & 0 & 8 \\ 2 & \sqrt{8} & 4 \end{pmatrix}.$$

With $Q = diag(-\sqrt{8}, 0, \sqrt{8})$ we have $C^{-1}C = I$ and $C^{-1}PC = Q$. Therefore, calculating $N^{(d)} = P^d N^{(0)}$ for $d \geq 1$ corresponds to $N^{(d)} = CQ^dC^{-1}N^{(0)}$, where $Q^d = \mathrm{diag}((-\sqrt{8})^d, 0, (\sqrt{8})^d)$. Hence, $N^{(d)}$ equals to

$$1/16 \begin{pmatrix} 2 & -2 & 2 \\ -\sqrt{8} & 0 & \sqrt{8} \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} (-\sqrt{8})^d & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & (\sqrt{8})^d \end{pmatrix} \begin{pmatrix} 2 & -\sqrt{8} & 4 \\ -2 & 0 & 8 \\ 2 & \sqrt{8} & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

which resolves to

$$N^{(d)} = 1/16 \left(4\sqrt{8}^d((-1)^d + 1), 2\sqrt{8}^{d+1}((-1)^{d+1} + 1), 2\sqrt{8}^d((-1)^d + 1)\right)^T.$$

The exact formula for $N^{(d)}$ and small values of d validates the observed search tree growth: $N^{(1)} = (0,2,0)^T$, $N^{(2)} = (4,0,2)^T$, $N^{(3)} = (0,16,0)^T$, $N^{(4)} = (32,0,16)^T$, etc.

The closed form for $N^{(d)}$ explicitly states that the asymptotic branching factor for the Eight Puzzle is $\sqrt{8}$. Moreover, the odd-even effect for branching in that puzzle is established by the factor $(-1)^d + 1$, which cancels for an odd value of d. Nevertheless, solving the characteristic equation and establishing the basis of eigenvectors by hand is tedious work. Fortunately, the application of symbolic mathematical tools such as Maple and Mathematica help to perform the calculations in larger systems.

For the Fifteen-Puzzle the basis-transformation matrix C and its inverse C^{-1} are

$$C = \begin{pmatrix} 1 & -1 & 1 \\ 1-\sqrt{5} & -1 & 1+\sqrt{5} \\ 3/2-1/2\sqrt{5} & 1 & 3/2+1/2\sqrt{5} \end{pmatrix}$$

and

$$C^{-1} = \begin{pmatrix} 1/50\ \left(5+3\sqrt{5}\right)\sqrt{5} & -1/50\ \left(5+\sqrt{5}\right)\sqrt{5} & 1/5 \\ -2/5 & -1/5 & 2/5 \\ 1/50\ \left(-5+3\sqrt{5}\right)\sqrt{5} & -1/50\ \left(-5+\sqrt{5}\right)\sqrt{5} & 1/5 \end{pmatrix}.$$

The vector of node counts is

$$N^{(d)} = \begin{pmatrix} 1/50\ \left(1-\sqrt{5}\right)^d \left(5+3\sqrt{5}\right)\sqrt{5} + 2/5 + \\ 1/50\ \left(1+\sqrt{5}\right)^d \left(-5+3\sqrt{5}\right)\sqrt{5} \\ 1/50\ \left(1-\sqrt{5}\right)\left(1-\sqrt{5}\right)^d \left(5+3\sqrt{5}\right)\sqrt{5} + 2/5 + \\ 1/50\ \left(1+\sqrt{5}\right)\left(1+\sqrt{5}\right)^d \left(-5+3\sqrt{5}\right)\sqrt{5} \\ 1/50\ \left(3/2-1/2\sqrt{5}\right)\left(1-\sqrt{5}\right)^d \left(5+3\sqrt{5}\right)\sqrt{5} - 2/5 + \\ 1/50\ \left(3/2+1/2\sqrt{5}\right)\left(1+\sqrt{5}\right)^d \left(-5+3\sqrt{5}\right)\sqrt{5} \end{pmatrix}$$

such that the exact total number of nodes in depth d is

$$1/50\ \left(7/2-3/2\sqrt{5}\right)\left(1-\sqrt{5}\right)^d \left(5+3\sqrt{5}\right)\sqrt{5} + 2/5 +$$

$$1/50\ \left(7/2+3/2\sqrt{5}\right)\left(1+\sqrt{5}\right)^d \left(-5+3\sqrt{5}\right)\sqrt{5}$$

The number of corner nodes (1,0,2,2,10,26,90,...), the number of side nodes (0,2,2,10,26,90,282,...) and the number of middle nodes (0,0,6,22,70,230,...) grow as expected. The largest eigenvalue $1+\sqrt{5}$ dominates the growth of the search tree in the limit for large d.

In the Twenty-Four-Puzzle the value $N^{(d)}$ equals

$$\begin{pmatrix} 1/36\ \left(-2\sqrt{3}\right)^d + 2/9\ \left(-\sqrt{3}\right)^d + 2/9\ \left(\sqrt{3}\right)^d + 1/36\ \left(2\sqrt{3}\right)^d \\ -1/18\sqrt{3}\left(-2\sqrt{3}\right)^d - 2/9\sqrt{3}\left(-\sqrt{3}\right)^d + 2/9\sqrt{3}\left(\sqrt{3}\right)^d + 1/18\sqrt{3}\left(2\sqrt{3}\right)^d \\ 1/18\ \left(-2\sqrt{3}\right)^d + 1/9\ \left(-\sqrt{3}\right)^d + 1/9\ \left(\sqrt{3}\right)^d + 1/18\ \left(2\sqrt{3}\right)^d \\ 1/12\ \left(-2\sqrt{3}\right)^d + 1/12\ \left(2\sqrt{3}\right)^d \\ -1/18\sqrt{3}\left(-2\sqrt{3}\right)^d + 1/9\sqrt{3}\left(-\sqrt{3}\right)^d - 1/9\sqrt{3}\left(\sqrt{3}\right)^d + 1/18\sqrt{3}\left(2\sqrt{3}\right)^d \\ 1/36\ \left(-2\sqrt{3}\right)^d - 1/9\ \left(-\sqrt{3}\right)^d - 1/9\ \left(\sqrt{3}\right)^d + 1/36\ \left(2\sqrt{3}\right)^d \end{pmatrix}$$

for the following total of nodes in depth d

$$n^{(d)} = 1/36\ \left(7-4\sqrt{3}\right)\left(-2\sqrt{3}\right)^d + 1/9\ \left(2-\sqrt{3}\right)\left(-\sqrt{3}\right)^d +$$

$$1/9\ \left(2+\sqrt{3}\right)\left(\sqrt{3}\right)^d + 1/36\ \left(7+4\sqrt{3}\right)\left(2\sqrt{3}\right)^d.$$

The value for small d validates that the total number of nodes increases as expected (2,6,18,60,198,684,...). Once again the vector of the largest absolute value determines the search tree growth.

If the size of the system is large, the exact value of $N^{(d)}$ has to be approximated. One option to bypass the intense calculations for determinants of large matrices and roots of high-degree polynomials is to compute the asymptotic branching factor b. The number of nodes in the brute-force search tree is then approximated by $n^{(d)} \approx b^d$.

6 Approximate Prediction of Search Tree Size

The matrix denotation for calculating the population of nodes according to the given equivalence relation implies $N^{(d)} = PN^{(d-1)}$, with $N^{(d)}$ being the vector of equivalent class sizes. The asymptotic branching factor b is given by the limit of $||N^{(d)}||_1/||N^{(d-1)}||_1$ which equals $N_i^{(d)}/N_i^{(d-1)}$ in any component $i \in \{1,\ldots,k\}$. Evaluating $N_i^{(d)}/N_i^{(d-1)}$ for increasing depth d is exactly what is considered in the algorithm of van Mises for approximating the largest eigenvalue (in absolute terms) of P. The algorithm is also referred to as the *power iteration* method.

As a precondition, the algorithm requires that P be diagonizable. This implies that we have n different eigenvalues $\lambda_1,\ldots,\lambda_n$ and each eigenvalue λ_i with multiplicity of α_i has α_i linear independent eigenvectors. Without loss of generality, we assume that the eigenvalues are given in decreasing order $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_k|$. The algorithm further requires that the start vector $N^{(0)}$ have a representation in the basis of eigenvectors in which no coefficient according to λ_1 is trivial.

We distinguish the following two cases: $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_k|$ and $|\lambda_1| = |\lambda_2| > \ldots \geq |\lambda_k|$. In the first case we obtain that (independent of the choice of

$j \in \{1, \ldots, k\}$) the value of $\lim_{d\to\infty} N_j^{(d)}/N_j^{(d-1)}$ equals $|\lambda_1|$. Similarly, in the second case $\lim_{d\to\infty} N_j^{(d)}/N_j^{(d-2)}$ is in fact λ_1^2. The cases $|\lambda_1| = \ldots = |\lambda_l| > \ldots \geq |\lambda_k|$ for $l > 2$ are dealt with analogously. The outcome of the algorithm and therefore the limit in the number of nodes in layers with difference l is $|\lambda_1|^l$, so that once more the geometric mean turns out to be $|\lambda_1|$.

We indicate the proof of the first case only. Diagonizability implies a basis of eigenvectors $b_1 \ldots, b_k$. Due to $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$ the quotient of $|\lambda_i/\lambda_1|^d$ converges to zero for large values of d. If the initial vector $N^{(0)}$ with respect to the eigenbasis is given as $x_1b_1 + x_2b_2 + \ldots + x_kb_k$ applying P^d yields $x_1P^db_1 + x_2P^db_2 + \ldots + x_kP^db_k$ by linearity of P, which further reduces to $x_1b_1\lambda_1^d + \lambda_2^dx_2b_2 + \ldots + \lambda_n^dx_kb_k$ by the definition of eigenvalues and eigenvectors. The term $x_1b_1\lambda_1^d$ will dominate the sum for increasing values of d. Factorizing λ_1^d in the numerator and λ_1^{d-1} in denominator of the quotient of $N_j^{(d)}/N_j^{(d-1)}$ results in an equation of the form $x_1b_1\lambda_1 + R$ where $\lim_{d\to\infty} R$ is bounded by a constant, since except of the leading term $x_1b_1\lambda_1$ both numerator and denominator in R only involve expressions of the form $O(|\lambda_i/\lambda_1|^d)$. Therefore, to find the asymptotic branching factor analytically, it suffices to determine the set of eigenvalues of P and to take the largest one. This corresponds to the results of the asymptotic branching factors in the (n^2-1)-Puzzles.

In the Eight-Puzzle the ratio $N_1^{(d)}/N_1^{(d-2)}$ is equal to 8 for $d > 3$ and, therefore, approximates λ_1^2. The value $n^{(d)}/\sqrt{8}^d$ alternates between 3/4 and $1/\sqrt{2}$. Hence, $\sqrt{8}^d$ approximates the search tree growth.

For the Fifteen-Puzzle for increasing depth d the value $N_1^{(d)}/N_1^{(d-1)}$ equals 1, 3, 13/5, 45/13, 47/15, 461/141, 1485/461, 4813/1485, 15565/4813, 50381/15565, 163021/50381, 527565/163021 = 3.236178161, etc., a sequence approximating $1+\sqrt{5} = 3.236067978$. Moreover, the ratio of $n^{(d)}$ and $(1+\sqrt{5})^d$ quickly converges to $1/50\ \left(7/2 + 3/2\sqrt{5}\right)\left(-5 + 3\sqrt{5}\right)\sqrt{5} = .5236067984$.

In the Twenty-Four-Puzzle the ratio $N_1^{(d)}/N_1^{(d-2)}$ converges to 12 starting with the sequence 6, 9, 11, 129/11, 513/43, 683/57, 8193/683, 32769/2731, 43691/3641 = 11.99972535, etc. The quotient $n^{(d)}/\sqrt{12}^d$ for larger depth alternates between .3888888889 and .3849001795 and is therefore bounded by a small constant.

If n is even – as in the Fifteen-Puzzle – the largest eigenvalue is unique and if n is odd – as in the Eight- and in the Twenty-Four-Puzzle – we find two eigenvalues with the same absolute value verifying that every two depths the node sizes will asymptotically increase by the square of these values.

7 Generalizing the Approach

Iterating the algorithm with $||N^{(d)}||_1/||N^{(d-1)}||_1$ instead of $N_j^{(d)}/N_j^{(d-1)}$ shows that the convergence conditions according to G and $\overline{G}$ are equivalent. This is important, since other graph properties may alter, e.g. symmetry of A_G is not inherited by $A_{\overline{G}}$. Therefore, we concentrate on diagonizability results of A_G,

which are easier to obtain. The *Theorem of Schur* states that symmetric matrices are indeed diagonizable. Moreover, the eigenvalues are real and the matrix to perform the basis transformation has the eigenvectors in its columns.

For the (n^2-1)-Puzzle we are done. Since G is undirected, A_G is indeed symmetric. In the spectrum of A_G power iteration either obtains a unique branching factor $b = |\lambda_1|$ or a branching factor of λ_1^2 for every two iterations. Therefore, the branching factor is the *spectral radius* $\rho = |\lambda_1|$.

7.1 Other Problem Spaces

Since the search tree is often exponentially larger than the problem graph we have reduced the prediction of the search tree growth to the spectral analysis of the explicit adjacency representation of the graph. As long as this graph is available, accurate and approximate predictions for the brute-force and subsequently for the heuristic search tree growth can be computed.

However, the calculations for large implicitly given graphs are involved such that reduction of the analysis to a smaller structure is desirable. For the (n^2-1)-Puzzle we proposed a compression to a few branching classes. The application of equivalence class reduction to exactly predict the search tree growth relies on the regular structure of the problem space. This technique is available as long as the same branching behavior for different states is given.

For *Rubik's Cube* without predecessor elimination $N^{(d)}$ equals 18^d since all nodes in the search tree span a complete 18-ary subtree. With predecessor elimination the node branching factor reduces to 15, since for each of the three twists *single clockwise*, *double clockwise*, and *counterclockwise* there is a remainder of five sides *front*, *back*, *right*, *left*, *up*, and *down* that are available. If we further restrict rotation of opposite sides to exactly one order we get the transition matrix $((6\ 6), (9\ 6))$, where the first class is the set of primary nodes with branching factor 15, and the second class is the class of secondary nodes with branching factor 12. The eigenvalues are $6 + 3\sqrt{6}$ and $6 - 3\sqrt{6}$ and the value $n^{(d)}$ equals $1/2\ \left(6 + 3\sqrt{6}\right)^d + 1/2\ \left(6 - 3\sqrt{6}\right)^d$. For small values of d experimental data as given in [11] matches this analytical study. The observed asymptotic branching factor is $6 + 3\sqrt{6} = 13.34846923$ as expected.

Extending the work to problem domains like the PSPACE-complete *Sokoban* problem [1] is challenging. It is difficult to derive an accurate prediction, since the branching behavior of the tree includes almost all state facets. Therefore, a more complicated search model has to be devised to derive exact or approximate search tree prediction in this domain. As Andreas Junghanns has coined in his Ph.D. dissertation [8], the impact of the search tree node prediction formula $\sum_{d=0}^{c} n^{(d)} P(c-d)$ has still to be shown. In the other PSPACE-complete sliding block game *Atomix* [7,6] simplification based on branching equivalences do apply and yield savings that are exponential in the number of atoms, but this void labeling scheme still results in an intractable size of the equivalence graph structure. Only very small games can be analyzed by this method.

7.2 Pruning

When incorporating pruning to the exploration process, symmetry of the underlying graph structure may be affected. Once again we consider the Eight-Puzzle. The adjacency matrix $A_{\overline{G}}^{pred}$ for predecessor elimination now consists of four classes: cs, sc, mc and cm, where the class ij indicates that the predecessor of a j-node in the search tree is an i node.

$$A_{\overline{G}}^{pred} = \begin{pmatrix} 0\,1\,0\,0 \\ 1\,0\,0\,1 \\ 2\,0\,0\,0 \\ 0\,0\,3\,0 \end{pmatrix}$$

In this case we cannot infer diagonizability according to the set of real numbers. Fortunately, we know that the branching factor is a positive real value since the iteration process is real. Therefore, we may perform all calculation to predict the search tree growth with complex numbers, for which the characteristic polynomial factorizes. The branching factor and the search tree growth can be calculated analytically and the iteration process eventually converges.

In the example, the set of (complex) eigenvalues is $i\sqrt{2}$, $-i\sqrt{2}$, $\sqrt{3}$, and $-\sqrt{3}$. Therefore, the asymptotic branching factor is $\sqrt{3}$. The vector $N^{(d)}$ is equal to

$$\begin{pmatrix} 1/5\,\left(i\sqrt{2}\right)^d + 1/5\,\left(-i\sqrt{2}\right)^d + 3/10\,\left(\sqrt{3}\right)^d + 3/10\,\left(-\sqrt{3}\right)^d \\ -1/10\,i\sqrt{2}\left(i\sqrt{2}\right)^d + 1/10\,i\sqrt{2}\left(-i\sqrt{2}\right)^d + 1/10\,\sqrt{3}\left(\sqrt{3}\right)^d - 1/10\,\sqrt{3}\left(-\sqrt{3}\right)^d \\ 3/20\,i\sqrt{2}\left(i\sqrt{2}\right)^d - 3/20\,i\sqrt{2}\left(-i\sqrt{2}\right)^d + 1/10\,\sqrt{3}\left(\sqrt{3}\right)^d - 1/10\,\sqrt{3}\left(-\sqrt{3}\right)^d \\ -1/10\,\left(i\sqrt{2}\right)^d - 1/10\,\left(-i\sqrt{2}\right)^d + 1/10\,\left(\sqrt{3}\right)^d + 1/10\,\left(-\sqrt{3}\right)^d \end{pmatrix}.$$

Finally, the total number of nodes in depth d is

$$n^{(d)} = 1/5\,\left(1/2 + 1/4\,i\sqrt{2}\right)\left(i\sqrt{2}\right)^d + 1/5\,\left(1/2 - 1/4\,i\sqrt{2}\right)\left(-i\sqrt{2}\right)^d +$$

$$1/10\,\left(4 + 2\,\sqrt{3}\right)\left(\sqrt{3}\right)^d + 1/10\,\left(4 - 2\,\sqrt{3}\right)\left(-\sqrt{3}\right)^d.$$

For small values of d the value $n^{(d)}$ equals 1, 2, 4, 8, 10, 20, 34, 68, 94, 188 etc.

7.3 Non-diagonizability

Since we assumed diagonizability, the eigenspaces $L(\lambda_i)$ according to the values λ_i have full rank α_i. In general this is not true. Not all matrices are diagonalizable. In this case the best thing one can do is to transform the matrix into *Jordan Form* which has blocks on the diagonal, each block being $r \times r$, with the eigenvalue on the diagonal, 1's above the diagonal and 0's everywhere else.

More precisely, a matrix A has *Jordan Form* J for an invertible matrix T, if $J = T^{-1}AT$ consists of so-called Jordan-blocks $J_1, \ldots, J_m$. One Jordan-block has an eigenvalue on the main diagonal and 1s on the diagonal above. Therefore, T gives a basis of eigenvectors and so-called *main vectors*. Each Jordan-block J_l of dimension j_l corresponds to one eigenvector t_1 and $j_l - 1$ main vectors $t_2, \ldots, t_{j_l}$ with $(A - \lambda_i I)t_0 = 0$ and $(A - \lambda_i I)t_m = t_{m-1}$, $m = 2, \ldots, j_l$. Using the Jordan basis one can devise $P^d N^{(0)}$ similar to the case above.

7.4 Start Vector

The second subtlety arises even if the matrix is diagonalizable. We are interested in determining the behavior of $P^d N^{(0)}$ for large d, where P is an $n \times n$ matrix and $N^{(0)}$ is an $n \times 1$ vector. Suppose that P is diagonalizable, which means that there is a basis of eigenvectors. Hence, $N^{(0)}$ can be written as a sum of eigenvectors: $N^{(0)} = v_1 + v_2 + v_3 + \ldots + v_n$ where v_i is an eigenvector with eigenvalue λ_i. It follows that $P^d N^{(0)} = \lambda_1^d v_1 + \lambda_2^d v_2 + \lambda_3^d v_3 + \ldots + \lambda_n^d v_n$ So the term with the largest corresponding eigenvalue will dominate for large d, *provided that the eigenvector is non-zero*. It may happen that the initial vector v has component of zero in the eigenspace of the largest eigenvalue. In general, the algorithm finds the largest eigenvalue in which the corresponding component is non-zero.

Fortunately, this observation is more theoretical in nature. In the iteration process this case is very rarely fulfilled. Rounding errors will soon or later lead to non-zero components. Moreover, to determine the asymptotic branching factor we have several initial states to choose from such that at least one has to yield non-zero coefficients.

8 Previous Work

This paper extends the work of Edelkamp and Korf [2] that already derived the asymptotic branching factors of the sliding-tile puzzles and Rubik's Cube. However, their approach lack sufficient convergence conditions. We established the criterion of diagonizability of the adjacency graph matrix of the problem graph that emerges of the algorithm of van Mises and showed that this criterion is fulfilled in undirected graphs by the Theorem of Schur. The (n^2-1)-Puzzles and Rubik's Cube are chosen to illustrate the techniques, since they are inherently difficult to solve and often considered in case studies.

The set of recurrence relations in [2] also showed that the numbers of nodes at various depths can be calculated in time linear to the product of the number of node classes and the depth of search tree by numerically iterating the recurrence relations. In contrast to this finding, the current paper resolves the problem of how to compute a closed form for the number of nodes. Last but not least, the given mathematical formalization of equivalence classification, diagonalization and power iteration builds a bridge for more powerful results in applying known mathematical theorems. At least in theory, generality to different problem spaces is given, since this approach applies to any problem graph with a diagonizable matrix and probably to more than that.

9 Conclusion and Discussion

In the paper we have improved the prediction of the number of node expansions in IDA* by an exact derivation of the number of nodes in the brute-force search tree. We have resolved the question of convergence to explain anomalies in of the asymptotic branching factor. The asymptotic branching factor is the spectral radius of the successor generation matrix and can be computed with the power iteration method. The approach extends to further regular problem spaces and can cope with simple pruning rules. The main result is that diagonizability is granted in undirected problem graphs, such that exact and approximate calculation of the brute-force search-tree are mathematically sound. The technique for establishing a closed form is not standard, and it is hard to suggest other methodologies to actually solve the set of recurrence relations.

Moreover, given the adjacency matrix P of an undirected graph by analyzing $N_i^{(d)}/N_i^{(d-1)}$ and $N_i^{(d)}/N_i^{(d-k)}$, $k > 1$, of the equation $N^{(d)} = N^{(d-1)}P$ gives the (mean) asymptotic branching factor. This is in fact the algorithm of van Mises to determine the largest eigenvalue of P for whose applicability we have to test if P is diagonizable. The paper closes the small gap in literature to accurately predict search tree growth in closed form and to compute the branching factor both analytically and numerically without relying on strong experimental assumption on the convergence.

Since for practical problems in which IDA* applies it is very unlikely that the entire graph structure can be kept in main memory, the approach helps only if some reduction of the branching behavior with respect to equivalence classes can be obtained. Therefore, the analysis is limited to the cases where the the successor generator matrix of the original or the adjacency graph structure can be build. If not, abstractions to the graph structure have to be found that preserve or approximate information of the branching behavior.

All analyses given in this or precursory papers on search tree prediction do not include the application of transposition tables, in which visited states together with their best encountered state merits (path length plus heuristic estimate) are kept. This in fact is also a challenge for analysts. One option is the prediction of the search tree growth of IDA* with respect to bit-state hashing, which turns out to be an improvement to transposition tables in single-agent games [7] and protocol verification [3]. For this model of partial search first results on coverage prediction have been found [4].

Exact calculation of the brute-force search tree raises the question if the other source of uncertainty, namely the heuristic equilibrium distribution, can also be eliminated. As said, the equilibrium distribution of the estimate can be obtained by random sampling. However, in some cases of regular search trees exact values can be produced. If the estimate is given with respect to a pattern database storing pairs of the form (estimated value, state pattern) by analyzing the pattern database, a histogram of heuristic values can computed: we determine the number of states that satisfy a pattern with a total to be computed for each integral heuristic value in a predefined range. For consistent heuristics

this range will be bounded by the heuristic estimate of the start state and the optimal solution length.

At the very far end of this research line there are precise or approximate predictions for the growth of A*'s and IDA*'s search efforts according to various kinds of heuristics, node caching strategies and problem domains. This implies an alternative way of defining heuristics themselves: ranking successor nodes according to the expected growth of the resulting search tree.

Acknowledgments. I thank Richard Korf and Michael Reid for intense and helpful discussions on the topic of this paper. The work is partially supported by DFG in the project *Heuristic Search and Its Application to Protocol Validation.* Thanks to T. Lauer for proof reading.

References

1. J. C. Culberson. Sokoban is PSPACE-complete. In *Fun for Algorithms (FUN)*, pages 65–76. Carleton Scientific, 1998.
2. S. Edelkamp and R. E. Korf. The branching factor of regular search spaces. In *National Conference on Artificial Intelligence (AAAI)*, 1998. 299–304.
3. S. Edelkamp, A. L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.
4. S. Edelkamp and U. Meyer. Theory and practice of time-space trade-offs in memory limited search. This volume.
5. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transaction on SSC*, 4:100–107, 1968.
6. M. Holzer and S. Schwoon. Assembling molecules in Atomix is hard. Technical Report 0101, Institut für Informatik, Technische Universität München, 2001.
7. F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to Atomix. This volume.
8. A. Junghanns. *Pushing the Limits: New Developments in Single-Agent Search.* PhD thesis, University of Alberta, 1999.
9. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
10. R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.
11. R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, pages 700–705, 1997.
12. R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 2001. To appear.
13. R. E. Korf and M. Reid. Complexity analysis of admissible heuristic search. In *National Conference on Artificial Intelligence (AAAI)*, 1998. 305–310.
14. R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of Iterative-Deepening-A*. *Artificial Intelligence*, 2001. To appear.
15. R. E. Korf and L. A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *National Conference on Artificial Intelligence (AAAI)*, pages 1202–1207, 1996.

Theory and Practice of Time-Space Trade-Offs in Memory Limited Search

Stefan Edelkamp[1] and Ulrich Meyer[2]

[1] Institut für Informatik
Georges-Köhler-Allee, Geb. 51
79110 Freiburg, Germany
edelkamp@informatik.uni-freiburg.de
[2] Max-Plank-Institut für Informatik
Stuhlsatzenhausenweg 85
66123 Saarbrücken, Germany
umeyer@mpi-sb.mpg.de

Abstract. Having to cope with memory limitations is an ubiquitous issue in heuristic search. We present theoretical and practical results on new variants for exploring state-space with respect to memory limitations.
We establish $O(\log n)$ minimum-space algorithms that omit both the open and the closed list to determine the shortest path between every two nodes and study the gap in between full memorization in a hash table and the information-theoretic lower bound. The proposed structure of suffix-lists elaborates on a concise binary representation of states by applying bit-state hashing techniques. Significantly more states can be stored while searching and inserting n items into suffix lists is still available in $O(n \log n)$ time. Bit-state hashing leads to the new paradigm of partial iterative-deepening heuristic search, in which full exploration is sacrificed for a better detection of duplicates in large search depth. We give first promising results in the application area of communication protocols.

1 Introduction

Heuristic search in large problem spaces inherently calls for algorithms capable of running under restricted memory. We present new data structures and algorithms that face the memory vs. duplication elimination problem that still arises even if the exploration is directed. The class of *memory-restricted search algorithms* has been developed under this aim. The framework imposes an absolute upper bound on the total memory the algorithm may use, regardless of the size of the problem space. If the number of nodes with distance value smaller than the optimal solution path length is larger than this memory bound, storing the entire list of visited nodes is no longer possible.

*Iterative deepening A**, IDA* for short [16], has proven effective to successively search the problem graph with bounded DFS traversals according to an increasing threshold for the tentative values. IDA* consumes space linear in the solution length. It does not use additionally available memory and traverses all generating paths. Unfortunately, the number of paths in a graph might be exponentially larger than the number of nodes such that the design of informative consistent heuristics and duplicate elimination remains

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 169–184, 2001.

Table 1. The IDA* Algorihm implemented with a Stack.

```
IDA*(s)
        Push(S, s, h(s)); U ← h(s)
        while (U ≠ ∞)
            U ← U'; U' ← ∞
            while (S ≠ ∅)
                (u, f(u)) ← Pop(S)
                if (goal(u)) return (u, f(u))
                for all v in Γ(u)
                        if (f(u) + w(u, v) − h(u) + h(v) > U)
                            if (f(u) + w(u, v) − h(u) + h(v) < U')
                                U' ← f(u) + w(u, v) − h(u) + h(v)
                        else
                            Push(S, v, f(u) + w(u, v) − h(u) + h(v))
```

essential. If all merits are distinct, IDA* expands a quadratic number of nodes in the worst case. Although iterative deepening is limited to small integral weights it performs well in practice. Table 1 depicts a possible implementation of IDA* in pseudo-code: S is a stack for backtracking, U is the current threshold, and U' the threshold for the next iteration. The value $w(u, v)$ is the weight of the transitition (u, v), $h(u)$ and $f(u)$ is the heuristic estimate and combined merit for node u, respectively.

Pattern data-bases [4] are a general tool to improve the estimate that can cope with complex subproblem interactions. A solution preserving relaxation of the search problem is traversed prior to the search and the goal distances of all abstract states are kept as lower bound estimats for the overall problem within a large hash table. However, the application of this pre-compilation technique is limited to suitable domain abstractions that yield better results than on-line computations as findings in protocol verification [8], AI-planning [6], and selected single-agent problems [14] indicate. Therefore, to lessen memory consumption according to a large number of states is still a problem.

Transposition tables are used to store and improve the distances until the memory bound has been reached [18]. However, when the memory is exhausted, IDA*'s time consumption is often stinged by uncaught duplicates.

Different node caching strategies have been applied: MREC [21] switches from A* to IDA* if the memory limit is reached. In contrast, SMA* [19] reassigns the space by dynamically deleting a previously expanded node, propagating up computed f-values to the parents in order to save re-computation as far as possible. However, the effect of node cashing is still limited. An adversary may request the nodes just deleted.

The paper is aimed to close this gap and is structured as follows: The first section gives an $O(\log n)$-space algorithm to search for the shortest path in graphs with uniform or small weights, with n being the total number of nodes in the problem graph. Suffix lists are a data structure for maximizing the number of stored states according to a given memory limit. The achieved result is compared to ordinary hashing and a derived information-theoretic bound. Bit-state state compaction, sequential hashing and partial search can substitute the transposition table of IDA* with a bit-vector table. Thereby, it is possible

Table 2. Computing the BFS Level.

```
Divide-And-Conquer-BFS(s)
  for i ← 1 to n
    for l ← 1 to n
      if (Path(s, i, l))
        print (s, i, l)
        break
```

```
Path(a, b, l)
  if ((a, b) ∈ E)
    return true
  else
    for j ← 1 to n
      if (Path(a, j, ⌈l/2⌉) and Path(j, b, ⌊l/2⌋))
        return true
    return false
```

to detect more duplicates in the space while increasing the depth of the search. We give promising experimental results in validating an industrial communication protocol.

2 Minimum Space Algorithms

First of all, we might ask for the limit of space reduction. Given a graph with n nodes we are interested in algorithms that compute the BFS-level and shortest paths of all nodes and either consume as little working space as possible or perform faster if more space is available. In addition, we assume that the algorithms are not allowed to modify the input during the exection.

The similar problems of node reachability (i.e., determine whether there any path between two nodes) and graph connectivity have been efficiently solved for the same restricted memory setting using random walk strategies [10,11]. However, we are not aware of any equivalent results for BFS and shortest paths. In the following we will devise an $O(\log n)$ space algorithm for BFS and shortest paths with small integer weights. The principle is similar to the simulation of nondeterministic Turing machines [20].

2.1 Divide-and-Conquer BFS

To compute the breadth-first-level for each node, with very limited space, we may use a DAC strategy *Path* that reports if there is a path from a to b with l edges. If l equals 1 and there is an edge from a to b then the procedure returns true. Otherwise, for each node index j, $1 \leq j \leq n$, we recursively determine *Path*$(a, j, \lceil l/2 \rceil)$ and *Path*$(j, b, \lfloor l/2 \rfloor)$. If both exist the returned value is true, compare Table 2. The recursion stack has to store at most $O(\log n)$ frames each of which contains $O(1)$ integers. Hence the space complexity is $O(\log n)$. However, this has to be paid with a time complexity of $O(n^{3+\log n})$ due to the recurrence equation $T(1) = 1$ and $T(l) = 2n \cdot T(l/2)$ resulting in $T(n) = (2n)^{\log n} = n^{1+\log n}$ time for one test. Varying b and iterating on l in the range of $\{1, \ldots, n\}$ gives the overall performance of $O(n^{3+\log n})$ steps.

2.2 Divide-and-Conquer SSSP

To extend this idea to the single-source shortest path problem (cf. Figure 3) with edge weights bounded by a constant C, we check the weights

Table 3. Searching the Shortest Paths.

```
Divide-And-Conquer-SSSP(s)
  for i ← 1 to n
    for w ← 1 to C · n
      if (Path(s, i, w))
        print (s, i, w)
        break
```

```
Path(a, b, w)
  if (weight(a, b) = w)
    return true
  else
    for j ← 1 to n
      for s ← max{1, ⌊w/2⌋ − ⌈C/2⌉}
            to min{w − 1, ⌈w/2⌉ + ⌈C/2⌉}
        if (Path(a, j, s) and Path(j, b, w − s))
          return true
    return false
```

$\lfloor w/2 \rfloor - \lceil C/2 \rceil$ for path 1,
$\lfloor w/2 \rfloor - \lceil C/2 + 1 \rceil$ for path 1,
...
$\lfloor w/2 \rfloor + \lceil C/2 \rceil$ for path 1,

$\lfloor w/2 \rfloor + \lceil C/2 \rceil$ for path 2,
$\lfloor w/2 \rfloor + \lceil C/2 \rceil - 1$ for path 2,
...
$\lfloor w/2 \rfloor - \lceil C/2 \rceil$ for path 2.

If there is a path with total weight w then it can be decomposed into one of above partitions. The worst-case reduction on weights is $Cn \to Cn/2 + C/2 \to Cn/4 + 3C/4 \to \ldots \to C \to C - 1 \to C - 2 \to C - 3 \to \ldots \to 1$. Therefore, the recursion depth is bounded by $\log(Cn) + C$ which results in a space requirement of $O(\log n)$ integers. As in the BFS case this compares to exponential time.

We do not claim practical applicability of these algorithms but want to make a start towards efficient shortest path algorithms for relatively little memory and unmodifiable large data, for example on optical read-only storage. In particular, time-space trade-offs seem to require new techniques.

3 Suffix Lists

Given m bits of memory, we want to maintain a dynamically evolving visited list *closed* under inserts and membership queries. The entries of *closed* are integers from $\{0, n\}$. Let r denote the maximal size of closed nodes that can be accommodated. As long as $n \leq m$ a simple bit array with bit i denoting element i is sufficient. Using hashing with open addressing, r is limited to $O(n/\log n)$. In the following we describe a simple but very space efficient approach with small update and query times. Similar ideas appeared in [2] but the data structure there is static and not theoretically analyzed. Another dynamic variant achieving asympotically equivalent storage bounds as our approach is sketched in [1]. However, constants are only given for two static examples. We provide constants for the dynamic version; comparing with the numbers of [1], our dynamic version could host up to five times more elements of the same value range. However, one has to take into consideration that the data structure of [1] provides constant access time whereas our structure incurs amortized logarithmic access time.

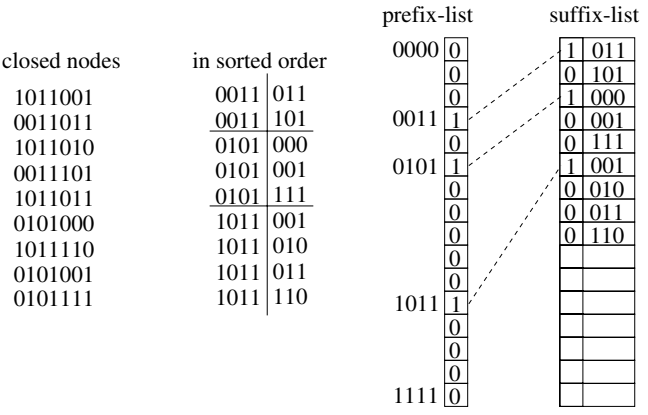


Fig. 1. Example for Suffix Lists with $p = 4$, and $s = 3$.

3.1 Representation

Let *bin*(u) be the binary representation of an element $u \leq n$ from the set *closed*. We split *bin*(u) in p high bits and $s = \lceil \log n \rceil - p$ low bits. Furthermore, $u_{s+p-1}, \ldots, u_s$ denotes the prefix of *bin*(u) and $u_{s-1}, \ldots, u_0$ stands for the suffix of *bin*(u).

A *suffix list data structure* consists of a linear array P of size 2^p bits and of a two-dimensional array L of size $r(m+1)$ bits. The basic idea of suffix lists is to store a common prefix of several entries as a single bit in P, whereas the distinctive suffixes form a group within L. P is stored as a bit array. L can hold several groups with each group consisting of a multiple of $s+1$ bits. The first bit of each $s+1$-bit row in L serves as a *group bit*. The first s bit suffix entry of a group has group bit one, the other elements of the group have group bit zero. We place the elements of a group together in lexicographical order, see Figure 1.

3.2 Searching

First, we compute $k = \sum_{i=0}^{p-1} u_{s+i} \cdot 2^i$ which gives us the search position in the prefix array P. Then we simply count the number of ones in P starting from position $P[0]$ until we reach $P[k]$. Let z be this number. Finally we search through L until we have found the zth suffix of L with group bit one. If we have to perform a membership query we simply search in this group. Note that searching a single entry may require scanning large areas of main memory.

3.3 Inserting

To insert entry u we first search the corresponding group as described above. In case u opens a new group within L this involves setting group bits in P and L. The suffix of u is inserted in its group while maintaining the elements of the group sorted. Note that an insert may need to shift many rows in L in order to create space at the desired position.

The maximum number r of elements that can be stored in S bits is limited as follows: We need 2^p bits for P and $s+1 = \lceil \log n \rceil - p + 1$ bits for each entry of L. Hence, we choose p so that r is maximal subject to

$$r \leq \frac{m - 2^p}{\lceil \log n \rceil - p + 1}.$$

For $p = \Theta(\log m - \log\log(n/m))$ the space requirement for both P and the suffixes in L is small enough to guarantee $r = \Theta\left(\frac{m}{\log(n/m)}\right)$.

3.4 Checkpoints

We now show how to speed up the operations. When searching or inserting an element u we have to compute z in order to find the correct group in L. Instead of scanning potentially large parts of P and L for each single query we maintain checkpoints, *one-counters*, in order to store the number of ones seen so far. Checkpoints are to lie close enough to support rapid search but must not consume more than a small fraction of the main memory. For $2^p \leq r$ we have $z \leq r$ for both arrays, so $\lceil \log r \rceil$ bits are sufficient for each one-counter.

Keeping one-counters after every $1/(c_1 \cdot \lfloor \log r \rfloor)$ entries limits the total space requirement. Binary search on the one-counters of P now reduces the scan-area to compute the correct value of z to $c_1 \cdot \lfloor \log r \rfloor$ bits.

Searching in L is slightly more difficult because groups could extend over 2^s entries, thus potentially spanning several one-counters with equal values. Nevertheless, finding the beginning and the end of large groups is possible within the stated bounds. As we keep the elements within a group sorted, another binary search on the actual entries is sufficient to locate the position in L.

3.5 Buffers

We now turn to insertions where two problems remain: adding a new element to a group may need shifting large amount of data. Also, after each insert the checkpoints must be updated. A simple solution uses a second buffer data structure *BU* which is less space efficient but supports rapid inserts and look-ups. When the number of elements in *BU* exceeds a certain threshold, *BU* is merged with the old suffix lists to obtain a new up-to-date space efficient representation. Choosing an appropriate size of *BU*, amortized analysis shows improved computational bounds for inserts while achieving asymptotically the same order of phases for the graph search algorithm.

Note that membership queries must be extended to *BU* as well. We implement *BU* as an array for hashing with open addressing. *BU* stores at most $c_2 \cdot r/\lceil \log n \rceil$ elements of size $p + s = \lceil \log n \rceil$, for some small constant c_2. As long as there is 10% space left in *BU*, we continue to insert elements into *BU* otherwise *BU* is sorted and the suffixes are moved from *BU* into the proper groups of L. The reason not to exploit the full hash table size is again to bound the expected search and insert time within *BU* to a constant number of tests.

Theorem 1. *Searching and inserting n items into suffix lists under space restriction m can be done in $O(n \cdot \log^2 n)$ bit operations. Assuming $\log n$ bits for a machine word, the total run time for n inserts and memberships is $O(n \log n)$.*

Proof. For a membership query we perform binary searches on numbers of $\lceil \log r \rceil$ bits or s bits, respectively. So, to search an element we need $O(\log^2 r + s^2) = O(\log^2 n)$ bit operations since $r \leq n$ and $s \leq \log n$.

Each of the $O(r/\log n)$ buffer entries consists of $O(\log n)$ bits, hence sorting the buffer can be done with

$$O\left(\log n \cdot \frac{r}{\log n} \cdot \log \frac{r}{\log n}\right) = O(r \cdot \log n)$$

bit operations. Starting with the biggest occurring keys merging can be performed in $O(1)$ memory scans, $O(m)$ operations. This also includes updating all one-counters. In spite of the additional data structures we still have

$$r = \Theta\left(\frac{m}{\log(n/m)}\right).$$

Thus, the total bit complexity for n inserts and membership queries is given by

$$\begin{aligned}
&O(\#\textit{buffer-runs}\ (\#\textit{sorting-ops} + \#\textit{merging-ops}) + \\
&\quad \#\textit{elements}\ \#\textit{buffer-search-ops} + \\
&\quad \#\textit{elements}\ \#\textit{membership-query-ops}) = \\
&O(n/r \cdot \log n \cdot (r \cdot \log n + m) + n \cdot \log^2 n + n \cdot \log^2 n) = \\
&O(n/r \cdot \log n \cdot (r \cdot \log n + r \cdot \log(n/m)) + n \cdot \log^2 n) = \\
&O(n \cdot \log^2 n).
\end{aligned}$$

Assuming a machine word length of $\log n$ in the RAM model, any modification or comparison of entries with $O(\log n)$ bits appearing in our suffix lists can be done using $O(1)$ machine operations. Hence the total complexity reduces to $O(n \cdot \log n)$ operations.

The constants can be improved using the following observation: in the case $n = (1+\epsilon) \cdot m$, for a small $\epsilon > 0$ nearly half of the entries in P will always be zero, namely those which are lexicographically bigger than the suffix of n itself. Cutting the P array at this position leaves more room for L which in turn enables us to keep more elements.

3.6 The Information Theoretic Bound

We place an upper bound on the maximal size r^* of the subset that can be stored. For the static case, we observe that $\lceil \log \binom{n}{r^*} \rceil \leq m$. However, if we consider the dynamic case, i.e. including insertions, we have to represent all former configurations. This results in

$$\left\lceil \log \left(\sum_{i=0}^{r^*} \binom{n}{i} \right) \right\rceil \leq m.$$

We aim choose r^* maximal subject to this inequality. For $r^* \leq (n-2)/3$ we have

$$\binom{n}{r^*} \leq \sum_{i=0}^{r^*} \binom{n}{i} \leq 2 \cdot \binom{n}{r^*}.$$

The correctness follows from $\binom{n}{i}/\binom{n}{i+1} \leq 1/2$ for $i \leq (n-2)/3$. We are only interested in the logarithms, so we conclude

$$\log \binom{n}{r^*} \leq \log \left(\sum_{i=0}^{r^*} \binom{n}{i} \right) \leq \log \left(2 \binom{n}{r^*} \right) = \log \binom{n}{r^*} + 1$$

Obviously in this restricted range it is sufficient to concentrate on the last binomial coefficient. The error in our estimate is at most one bit. The restriction on r^* is compatible with all reasonable choices for n and m. Using

$$\begin{aligned} \log \binom{n}{r^*} &= \log \frac{n \cdot (n-1) \cdot \cdots \cdot (n-r^*+1)}{r^*!} \\ &= \sum_{j=n-r^*+1}^{n} \log j - \sum_{j=1}^{r^*} \log j, \end{aligned}$$

we can approximate the logarithm by two corresponding integrals. If we properly bias the integral limits we can be sure to compute a lower bound

$$\log \binom{n}{r^*} \geq \int_{n-r^*+1}^{n} \log(x)\, \mathrm{d}x - \int_{2}^{r^*+1} \log(x)\, \mathrm{d}x.$$

Maximizing r^* with respect to this equation yields an information theoretic upper bound.

Table 4 compares suffix lists with hashing and open addressing. The constants for suffix lists are chosen so that $2 \cdot c_1 + c_2 \leq 1/10$ which means that if r elements can be treated, we set aside $r/10$ bits to speed-up internal computations. For hashing with open addressing we also leave 10% memory free to keep the internal computation time moderate. When using suffix lists instead of hashing, note that only the ratio between n and m is important. For the static data structure of [1] the following numbers are given: for $\frac{n}{m} = \frac{1.0 \cdot 2^{32}}{1.9 \cdot 2^{30}} \approx 1.05$ it can store a fraction of $\frac{r}{n} = \frac{1.4 \cdot 2^{27}}{1.0 \cdot 2^{32}} \approx 4.37\%$ of n. Our approach achieves 22.7% which constitues an improvement by a factor of more than five. For another example with $n/m \approx 3.2$ our approach gains by a factor of about 1.8.

Hence, suffix lists can close the phase gap in search algorithms between the upper bound and trivial approaches like hashing with open addressing. Already for $n \geq 1.1 \cdot m$ we reach two-optimality.

4 Bit-State Hash-Tables

Advanced to the treatment of data structures and algorithms we give a small introduction to the verification of distributed software systems and communication protocols; an apparent and practical relevant domain for state-space search.

Table 4. Fractions of n stored in Suffix Lists and Hashing with Open Addressing.

n/m	Upper Bound	Suffix Lists	Hashing $n = 2^{20}$	Hashing $n = 2^{30}$
1.05	33.2 %	22.7 %	4.3 %	2.9 %
1.10	32.4 %	21.2 %	4.1 %	2.8 %
1.25	24.3 %	17.7 %	3.6 %	2.4 %
1.50	17.4 %	13.4 %	3.0 %	2.0 %
2.00	11.0 %	9.1 %	2.3 %	1.5 %
3.00	6.1 %	5.3 %	1.5 %	1.0 %
4.00	4.1 %	3.7 %	1.1 %	0.7 %
8.00	1.7 %	1.5 %	0.5 %	0.4 %
16.00	0.7 %	0.7 %	0.3 %	0.2 %

4.1 State Space Search for Protocols Validation

Reliable communication is probably the most important issue for accessing the Internet and for the design of distributed computer systems. Usually a layered structure like the ISO Reference Model is used to allow for different abstractions. In one layer (transport layer) we have the request for reliable communication while the next lower layers provide this quality of service facing a lossy channel (cf. Figure 2).

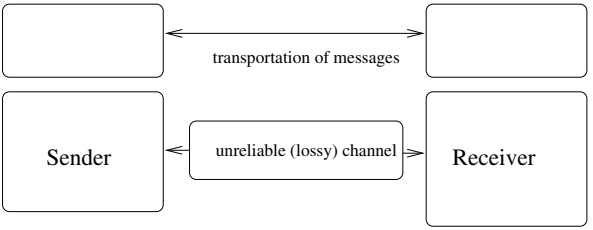


Fig. 2. Communication over a Lossy Channel for Messaging in Layered Protocols.

One example to cope with lossy channels is the alternating bit protocol. The message flow is visualized in Figure 3. To assert secure data transport from the sender to the receiver we assume sequence numbers for messages. In the following we study algorithms and data structures to certify the correctness of a such a protocol.

4.2 Supertrace

The idea of bit-state hashing is adopted from Holzman's protocol validator Spin [12], that parses the expressive concurrent Promela protocol specification language. It compresses the state description of several hundred bits down to only a few bits to build a hash table with up to 2^{30} entries and more. Combined with a depth-first search strategy this is in

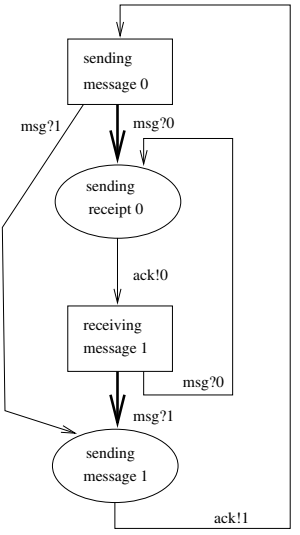


Fig. 3. Flow of Control on a Lossy Channel with the Alternating Bit Protocol.

fact the *supertrace algorithm*: A state s is represented by its hash address $h(s)$. When generating a state the corresponding bit is set. Synonyms are regarded as duplicates resulting in pruning the search. The search algorithm is not complete, since not all synonyms are disambiguated. Moreover, through depth-first traversal, the length of a witness for an encountered error state is not minimal.

4.3 Data Structures

As an illustration and generalization of the bit-state hashing idea, Figure 4 depicts the range of possible hash structures: Usual hashing with chaining of synonyms, single-bit hashing, double-bit hashing and hash compact [22]. Let n be the number of reachable states and m be the maximal number of bits available. A coarse approximation for single bit-state hashing coverage with $n < m$ is $1 - P_1$ with the average probability of collision $P_1 \leq \frac{1}{n} \sum_{i=0}^{n-1} \frac{i}{m} \leq n/2m$, since the i-th element collides with one of the $i-1$ already inserted elements with a probability of at most $(i-1)/m$, $1 \leq i \leq n$ [13]. For multi-bit hashing and h (independent) hash-functions by assuming $hn < m$ coverage is improved to $1 - P_h$ with average probability of collision $P_h \leq \frac{1}{n} \sum_{i=0}^{n-1} (h \cdot \frac{i}{m})^h$, since i elements occupy at most hi/m addresses, $0 \leq i \leq n-1$. For double bit-state hashing this simplifies to $P_2 \leq \frac{1}{n} (\frac{2}{m})^2 \sum_{i=0}^{n-1} i^2 = 2(n-1)(2n-1)/3m^2 \leq 4n^2/3m^2$.

4.4 Sequential and Universal Hashing

The drawback in incompleteness of partial search is compensated by re-invoking the algorithm with different hash functions to improve the coverage of the search tree.

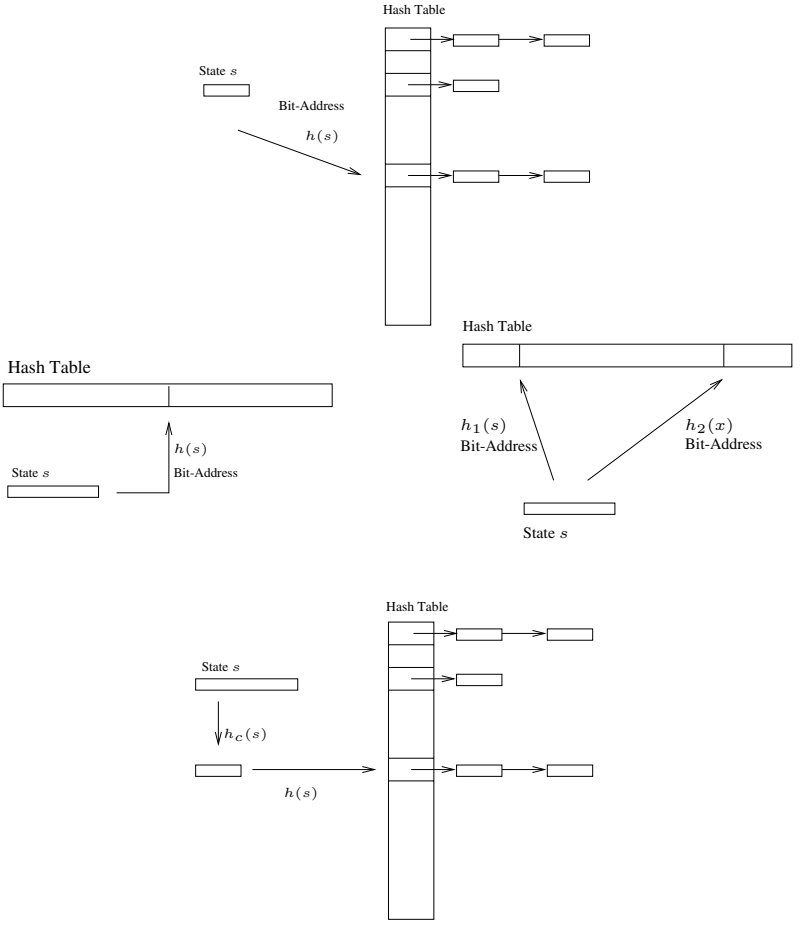


Fig. 4. Ordinary Hashing, Single Bit-State Hashing, Double Bit-State Hashing, and Hash-Compact.

Subsequently, this technique, called *sequential hashing*, examines various beams in the search tree (up to a certain threshold depth). In considerably large protocols supertrace with sequential hashing succeeds in finding bugs but still returns long witness trails. If in sequential hashing exploration with the first hash first function covers m/n of the search space, the probability that a state x is not generated in d independent runs is $(1-m/n)^d$ such that x is reached with probability $1-(1-m/n)^d$. Eckerle and Lais [5] have shown that this *ideal* circumstances are not given in practice and refine the model for coverage prediction.

Moreover, universal hash functions suit best for implementing sequential hashing. Let A, B be sets with $|B| = 2^w$, for some integer value w. The class of hash functions $\mathcal{H}$ is *universal*, if for all $x, y \in A$, we have

$$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{|B|}.$$

Universal hash functions lead to a good distribution of values on the average. If h is drawn randomly from $\mathcal{H}$ and S is the set of keys to be inserted in the hash table, the expected cost of each search, insert and delete operation is bounded by $(1 + |S|/|B|)$. We give an example of a universal hash function. Let p be prime, and $p \geq |A|$ and $h_{m,n}(s) = ((m \cdot s + n) \bmod p) \bmod |B|$. Then the class $\mathcal{H}_1 := \{h_{m,n} | m, n \in Z_p\}$ is universal.

4.5 Validating Process

For the validation of the design of the protocols, bug-finding by simulation and testing has its drawbacks, since several subtle bugs in concurrent systems are difficult to establish. Given a formal specification of a desired protocol property model-checking is a push-button procedure to verify the correctness. Validation is performed by traversing the finite-state machine representation of the protocol to find a bug. Therefore protocols are represented by state spaces, in which reachability analysis is performed to establish error states.

Therefore, directed search for minimal counterexamples in the protocol space according to a given implementation corresponds to the search for an optimal solution with the goal as the failure state. From a model checking perspective [3] the approach allows to implement various heuristics to direct the search into the direction of the failure. From an AI-perspective partial search, maybe assisted with sequential hashing, condenses duplicate information in various search and planning problem spaces.

4.6 Heuristic Search Algorithm

The apparent aspirant for state compaction is IDA* with *transposition tables*, since, in opposite to A*, it tracks the solution path on the stack, which allows to omit the predecessor link in the state description of the set of visited states.

When substituting the transposition table H of already visited nodes in IDA* by bit-state, multi bit-state or hash compaction we establish the Partial IDA* algorithm as depicted in Table 5. Since neither the predecessor nor the f-value are present, in order to distinguish the current iteration from the previous ones, the bit-state table has to be re-initialized in each iteration of IDA*. Refreshing large bit-vector tables is fast in practice, but for shallow searches with a small number of expanded nodes this scheme can be improved by invoking ordinary IDA* with transposition table updates for smaller thresholds and by applying bit-vector exploration in large depths only.

In practice the obtained counterexamples are minimal, since the coverage with bit-state duplicate elimination is very close to 100 % for moderately sized systems ($n < m$). Moreover, the technique of *trail-directed search* can effectivly improve non-optimal existing paths [9].

The results for searching deadlocks in one large communication protocol are depicted in Table 6, where the number of expansions with respect to different optimal search algorithms for an increasing threshold is shown. For A* a snapshot is taken at each time the priority queue value increases, while in IDA* the number of expanded nodes according to each completed iteration is shown. Hence, the number of node expansion numbers in these two algorithms do not match exactly, but indicate a common trend.

Table 5. Partial IDA* Algorithm.

```
Partial IDA*(s)
    Push(S, s, h(s)); U' ← U ← h(s)
    while (U ≠ ∞)
        U ← U'; U' ← ∞
        Init(H)
        while (S ≠ ∅)
            (u, f(u)) ← Pop(S)
            if (goal(u)) return (u, f(u))
            for all v in Γ(u)
                if (Search(H, v) ≠ ∅)
                    Insert(H, v)
                    if (f(u) + w(u, v) − h(u) + h(v) > U)
                        if (f(u) + w(u, v) − h(u) + h(v) < U')
                            U' ← f(u) + w(u, v) − h(u) + h(v)
                    else
                        Push(S, v, f(u) + w(u, v) − h(u) + h(v))
```

The considered protocol instance is the industrial General Inter-ORB Protocol (GIOP, 1 server and 3 clients) [15], which is a key component of the Common Object Request Broker Architecture (CORBA) specification.

The witness for a seeded deadlock in depth 70 has to be established according to the heuristic that counts the number of non-active processes. The state vector generated by the validator tool SPIN is 544 Bytes large, such that the visited list (hash table or transposition table) is bounded to 2^{18} states corresponding to approx. 2^{17} KByte or 128 MByte. Therefore, we fix the size of the bit-state hash table accordingly at 2^{30} Bits.

Algorithm A* exceeds its space limit in depth 61 and fails. IDA* utilizes a transposition table which is exhausted at the same depth. As IDA* then searches the tree of generation paths it compensates space for time. But even when investing more than 24 hours on our 248 MHz Sun Ultra Workstation and when utilizing the table constructed so far, ordinary IDA* was not able to complete search depth 61. On the other hand, Partial A* finishes all searches up to depth 70 with either single- and double bit-state hashing within a total of one hour.

Since the algorithms are not complete, we validated optimality with A* with our maximum of 1.5 GByte main memory. Note that the difference in the number of node expansions in single and double bit-state hashing is very small (less than a hundred) and only occurs in large search depths (iteration 58 onwards). As Partial IDA* with double bit-state hashing expands exactly the same number of states as IDA* with a transposition table, we actually observe no loss of information in the example.

5 Conclusion

At the limit of main memory eliminating duplicates and weight diversity can soon result in thrashing both resources time and space, such that powerful data structures for caching,

Table 6. Number of Expanded nodes of Search Algorithms in the GIOP Protocol.

depth	A* (hash table)	IDA* (transposition table)	Partial IDA* (single bit-state)	Partial IDA* (double bit-state)
⋮	⋮	⋮	⋮	⋮
40	6,646	6,333	6,333	6,333
41	9,306	8,184	8,184	8,184
42	10,955	10,575	10,575	10,575
43	13,666	13,290	13,290	13,290
44	17,761	16,500	16,500	16,500
45	20,130	19,860	19,860	19,860
46	25,426	23,646	23,646	23,646
47	27,714	27,654	27,654	27,654
48	33,799	32,040	32,040	32,040
49	37,095	37,011	37,011	37,011
50	46,105	42,849	42,849	42,849
51	51,113	49,872	49,872	49,872
52	61,710	58,545	58,545	58,545
53	73,195	69,162	69,162	69,162
54	85,245	81,993	81,993	81,993
55	96,995	96,543	96,543	96,543
56	113,950	112,296	112,296	112,296
57	115,460	129,138	129,138	129,138
58	147,042	146,625	146,623	146,625
59	150,344	164,982	164,978	164,982
60	184,872	184,383	184,376	184,383
61	187,411	206,145	206,135	206,145
62	-	$>$ 97,157,721	229,611	229,626
63	-	-	255,386	255,411
64	-	-	282,416	282,444
65	-	-	311,306	311,340
66	-	-	341,522	341,562
67	-	-	373,374	373,422
68	-	-	407,249	407,310
69	-	-	442,863	442,941
70	-	-	67	67

partial search and compressed dictionaries are required. Therefore, regarding the limits and possibilities of A*, we have suggested different contributions to memory-restricted search. Partial search supports bookkeeping in tremendously large hash tables to avoid duplicates in the search, while suffix lists push the envelope for increasing the number of nodes to be stored without loss of information.

The treatment of Partial IDA* search elaborates on precursoring findings in [8], where a rudimentory bit vector and single-bit hashing function has been chosen for implementation. For the experiments we chose a non-trivial protocol example [7], but recent progress shows that the algorithm has also reduced the search efforts for optimally

solving Atomix, a PSPACE-complete AI single-agent search problem [14]. Omitting the visited list and exploring the space in a Divide-and-Conquer fashion has been proposed in [17], and the algorithms we consider study the effect of removing the horizon-list as well. Another model checking approach for state compression as to answer to the representation problem of large sets of states are binary decision diagrams (BDDs) that are able to encode large sets of states without necessarily encountering exponential growth. However, hybrid methods of explicite and symbolic search methods are still to be developed.

Acknowledgements. The authors would like to thank Robert Holte for valuable discussions and comments and Alberto Lluch Lafuente for the joined implementation efforts in the HSF-SPIN protocol validator. The work is partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by DFG within the project *Heuristic Search and Its Application to Protocol Validation.*

References

1. A. Brodnik and J. Munro. Membership in constant time and almost-minimum space. *SIAM*, 28(3):1627–1640, 1999.
2. Y. Choueka, A. Fraenkel, S. Klein, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. pages 88–96, 1986.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
4. J. C. Culberson and J. Schaeffer. Searching with pattern databases. Lecture Notes in Computer Science, pages 402–416. Springer, 1996.
5. J. Eckerle and T. Lais. Limits and possibilities of sequential hashing with supertrace. In *IFIP FORTE/PSTV*, Lecture Notes in Computer Science. Springer, 1998.
6. S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, 2001. To appear.
7. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *SPIN Workshop*, Lecture Notes in Computer Science, pages 57–79. Springer, 2001.
8. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.
9. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Trail-directed model checking. In *Workshop on Software Model Checking*. Electronic Notes in Theoretical Computer Science, Elsevier, 2001. To appear.
10. U. Feige. A fast randomized LOGSPACE algorithm for graph connectivity. *Theoretical Computer Science*, 169(2):147–160, 1996.
11. U. Feige. A spectrum of time-space tradeoffs for undirected $s - t$ connectivity. *Journal of Computer and System Sciences*, 54(2):305–316, 1997.
12. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
13. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design: An International Journal*, 13(3):289–307, 1998.
14. F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier. Finding optimal solutions to Atomix. This volume.
15. M. Kamel and S. Leue. Formalization and validation of the general inter-orb protocol (GIOP) using Promela and SPIN. In *Software Tools for Technology Transfer*, volume 2, pages 394–409, 2000.

16. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
17. R. E. Korf. Divide-and-conquer bidirectional search: First results. In *IJCAI*, pages 1184–1189, 1999.
18. A. Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.
19. S. Russell. Efficient memory-bounded search methods. In *ECAI-92*, pages 1–5, 1992.
20. W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.
21. A. K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, pages 297–302, 1989.
22. U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90. Shaker Verlag, Aachen, 1996.

Hierarchical Diagnosis of Large Configurator Knowledge Bases

Alexander Felfernig[1], Gerhard E. Friedrich[1], Dietmar Jannach[1], Markus Stumptner[2], and Markus Zanker[1]

[1] Computer Science and Manufacturing Research Group, University of Klagenfurt, Austria
{felfernig, friedrich, jannach, zanker}@ifit.uni-klu.ac.at

[2] Database and Artificial Intelligence Group, Technische Universität Wien
mst@dbai.tuwien.ac.at

Abstract. Debugging, validation, and maintenance of configurator knowledge bases are important tasks for the successful deployment of product configuration systems. Consistency-based diagnosis has shown to be a promising approach for detecting faulty parts in the knowledge bases and explaining unexpected behavior of the configurator, whereby (partial) configurations are used as test cases. In this paper we show how hierarchical diagnosis can be employed to cope with the complexity of debugging large configurator knowledge bases. A framework for hierarchical diagnosis on different levels of abstraction is presented as well as an algorithm for the calculation of diagnoses on those levels. The presented approach aims at the reuse of existing special purpose configuration systems. We show that the exploitation of hierarchies in such problem domains leads to a significant efficiency enhancement thus broadening the applicability of consistency-based diagnosis.

1 Introduction

Knowledge-based configuration is an important application field of AI technology due to increasing demand for configurable and mass-customized products. The increased complexity and high change rates of the products (and the corresponding knowledge bases) made adequate debugging and testing support a prerequisite for successful deployment of such tools. Consistency-based diagnosis techniques have shown to be applicable not only for diagnosing electronic circuits or other hardware devices, but also for debugging of software systems such as logic programs [4], repairing inconsistencies in databases [10] or VHDL programs [9].

In [5] it was shown how model-based diagnosis (*MBD*) can also be employed for error detection within knowledge bases for configuration systems. A framework is described where "positive" and "negative" examples can be provided, where (partial or complete) "positive" examples should be accepted or completed by the configurator and "negative" ones should be rejected. The behavior of the configurator is unexpected, if a positive example cannot be completed or causes a contradiction or an intended negative example is falsely accepted. For localizing possible explanations, the problem is mapped to a diagnosis problem, where the sentences

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 185–197, 2001.

(constraints) of the knowledge base play the part of *components* from the MBD terminology.

The success of AI techniques in the configuration area is based on highly specialized configurators ([7], [13]) able to deal with large configuration problems. In addition such systems use a restricted first-order logical language. Our goal is to extend such specialized systems with diagnosis capabilities for debugging. The work of [16] shows how such an integration of special consistency-checking systems into a general diagnosis problem solver can be achieved. However, the efficiency of the diagnosis computation depends heavily on the minimality of the conflict sets. Typically, specialized configurators offer no generation of conflict sets or provide non-minimal conflicts including many additional elements. One reason for this is that in configuration we have to deal with general clauses and many dependencies between these constraints. Therefore, dependency tracing on top of constraint propagation will not reduce the conflicts significantly in practice.

In this paper we show that hierarchical abstraction ([1],[12],[15],[17]) significantly enhances the efficiency of diagnosing configuration knowledge bases. Typical hierarchical structures in configurator knowledge bases can be employed for diagnosis at different levels of granularity. The paper is organized as follows. After giving a motivating example, we shortly review the framework of consistency-based diagnosis of configurator knowledge bases. We present the extension of this approach with structural abstractions for diagnosis at different levels. After the description of an algorithm and first results from a prototypical implementation, we discuss related work and present our conclusions.

2 Motivating Example

For demonstration purposes we use a small fragment of a configuration problem. After inserting a typical failure, we show how consistency-based diagnosis can help to detect this error. We use first order sentences to ensure clear representation with precise semantics. In the example, we have a frame, where cards of different types (CPUs and switching modules) can be inserted on existing named slots. The knowledge-base consists of the following definitions, where *types* describes the available component types and *ports* describes the predefined connection points for the components [14]:

types = {frame, cpu-1, cpu-2, sm-1, sm-2};
ports(frame) = {cp1,cp2,smp1,smp2}.
ports(cpu-1) = {framep}. ports(cpu-2) = {framep}.
ports(sm-1) = {framep}. ports(sm-2) = {framep}.

We use the following predicates for describing configuration knowledge: *type(c,t)* associates a component identifier with one of the predefined types, *conn(c1,p1,c2,p2)* describes that component instance *c1* is connected on port *p1* with another component *c2* on *p2*. Attribute valuations of components are described by a predicate *val/3* which is omitted here (see 5 for an example). In addition, definitions are contained describing which components can be connected via which ports in general and other domain-independent constraints, like that one port can be connected to exactly one

other port and connections are symmetric. In addition, the following constraints for the individual component types have to hold:

*"**CtF$_1$**: If there is a cpu-1 on cp2, there must be a sm-1 on one of the switching module ports.",* more formally

$$\forall C,F : type(F,frame) \wedge type(C,cpu\text{-}1) \wedge conn(F,cp2,C,framep) \Rightarrow \exists (S,P): type(S,sm\text{-}1) \wedge conn(F,P,S,framep) \wedge P \in \{smp1, smp2\}.$$

For sake of brevity, we only describe the other constraints without formal representation.

*"**CtS$_1$**: If there is a switching-module sm-2 on smp1 there must be also one sm-2 on smp2."*

*"**CtC$_1$**: If there is a CPU of any type connected to any CPU port, at least one switching module of type sm-1 or sm-2 must be connected to smp1 or smp2."*

*"**CtC$_2$**: A CPU of type cpu-2 on port cp1 requires switching modules of type sm-2 on both ports smp1, smp2."*

*"**CtC$_3$**:A CPU of type cpu-1 on cp2 requires a CPU of the type cpu-2 on cp1."*

Let us assume, that CtF_1 is faulty and too restrictive, because also switching modules of type *sm-2* should be allowed. This situation came about because *sm-2* was a type newly introduced to the knowledge base and CtF_1 was not maintained correctly. The user provides a positive example with one *cpu-1* and a switching module *sm-2*:

$$e^+ = \{\exists F,S,C: type(F,frame) \wedge type(S,sm\text{-}2) \wedge type(C,cpu\text{-}1) \wedge conn(F,cp\text{-}2,C,framep) \wedge conn(F, smp\text{-}1, S, framep)\}.$$

Note, that the partial example cannot be completed to a correct configuration (See [5] for the usage of negative examples). Following the consistency-based approach from [5] the minimal conflicts [16] (sets of constraints causing a contradiction with the example)

$\{CtF_1, CtS_1\}$ and $\{CtF_1, CtC_3, CtC_2\}$

induce the minimal diagnoses for the unexpected behavior:

$\{ab(CtF_1).\}$ $\{ab(CtS_1).\ ab(CtC_3).\}$ $\{ab(CtS_1).\ ab(CtC_2).\}$

In other words, if these sets of constraints (diagnoses) are considered abnormal and are canceled, the partial configuration can be completed by the configurator.

Note, that CtC_1 is not contained in any minimal conflict set and that the assumption of having minimal conflict sets available for the HS-DAG ([11],[16]) generation is very strong for the configuration domain for different reasons: First, there are many interdependencies between the constraints, so a dependency analysis or tracing will not suffice. Second, for the configuration domain, tools (e.g., [12]) with specialized inference mechanisms (*Generative Constraint Satisfaction* [7]) and limited explanation facilities (and costly conflict minimization) are employed for effective calculation of configurations. In addition, the constraint language must be expressive enough for the domain, i.e., the language's constructs exceed the expressiveness of *Horn*-clauses. Consequently, the conflicts returned by the specialized *theorem prover*

are large and non-minimal, leading to high search complexity during the HS-DAG generation, because overlaps in the conflict sets minimize benefits from techniques like conflict reuse. One could argue that we should minimize the conflicts before using them during HS-DAG generation. However, note that the computation of additional minimal conflicts is costly, because conflicts with many redundant elements may contain several minimal conflicts, such that minimization requires a number of consistency checks, which is not linear in the number of elements of the non-minimal conflict.

Following a hierarchical approach, we will try to analyze the system on a coarse level with smaller complexity, i.e., we do not diagnose individual constraints but whole groups of related constraints. If we group the constraints (indicated by the names in the example) and assume all contained constraints to be faulty if such a named group is "abnormal", the minimal diagnoses will be

{ab(frame).} {ab(sm). ab(cpu).}

A faulty constraint is simply not considered for consistency checking, i.e., we do not consider fault models.

Having only three diagnosable components at the abstract level (i.e., the constraint groups *frame, sm,* and *cpu*), this result can be calculated very fast. Then, the user can decide to use the result as a pointer to a faulty group or refine the diagnoses to the next level. When refining one diagnosis to the next level, the contained faulty groups are replaced by their elements; groups that were not assumed to be faulty are still treated as one component, which reduces the number of diagnosable components.

If we extend the example to have five component types (groups), each containing ten constraints (diagnosable components *n=50*) and the diagnoses as before, the number of theoretically possible combinations for double-faults is $\binom{50}{2} = 1.225$.

This number is not reduced significantly when only large non-minimal conflicts are available. Using the hierarchical approach, for the top-level diagnosis there are only ten possible two-element combinations for five groups. Refining the first abstract diagnosis with replacement of the faulty group *frame* leads to *n=10+4* diagnosable components and *n=20+3* for the second diagnosis. So, the theoretical upper limit for hierarchical approach for double faults without regarding conflicts is $\binom{5}{2}+\binom{14}{2}+\binom{23}{2} = 344.$

3 Diagnosing Configurator Knowledge Bases

In our general framework, a configurator knowledge base consists of a set of logical sentences *DD* describing available component types, their attributes and connection points as well as constraints on legal product constellations [14]. Configuration problems are solved according to specific user requirements *SRS*. A configuration result can be described by means of a set of ground literals containing information on component instances, attribute values and connections. The set of possible literals is contained in a set *CONL*.

Definition: (Configuration problem): *A configuration problem is described by a triple (DD,SRS,CONL), where DD and SRS are sets of logical sentences and CONL is a set of predicate symbols.*
DD represents the domain description, SRS the user requirements for a configuration problem instance. A configuration CONF is described by a set of ground literals whose predicate symbols are in CONL. ∘

Definition (Consistent configuration): *Given a configuration problem (DD,SRS, CONL), a configuration CONF is consistent iff* $DD \cup SRS \cup CONF$ *is satisfiable.* ∘

To ensure the completeness of a configuration, additional formulae for each symbol in *CONL* have to be introduced to *CONF*, e.g., $type(X,Y) \Rightarrow (type(X,Y) \in CONF)$.

We denote the configuration *CONF* extended by these axioms with $\overline{CONF}$. (For a detailed exposition, see 5).

Definition (Valid and irreducible configuration): *Let (DD,SRS,CONL) be a configuration problem. A configuration CONF is valid iff* $DD \cup SRS \cup \overline{CONF}$ *is satisfiable. CONF is irreducible if there exists no other valid configuration* $CONF^{sub}$ *such that* $CONF^{sub} \subset CONF$. ❑

Definition (CKB-Diagnosis Problem): *A CKB (Configuration Knowledge Base) Diagnosis Problem is a triple* (DD,E^+,E^-) *where DD is a configuration knowledge base,* E^+ *is a set of positive and* E^- *a set of negative examples given as sets of logical sentences. We assume each example on its own to be consistent.* ❑

Positive examples are (partial) configurations, which should be accepted by the configurator, whereas negative examples should be rejected. Given these example sets and the domain description cause an inconsistency, a diagnosis corresponds to the removal of possibly faulty sentences restoring the consistency. In addition, if a negative example is consistent with the knowledge base, we have to find an extension to *DD* which restores inconsistency for all such negative examples.

Definition (CKB-Diagnosis): *A CKB-Diagnosis for a CKB-Diagnosis Problem (DD,* E^+,E^-*) is a set* $S \subseteq DD$ *such that there exists an extension EX, where EX is a set of logical sentences, such that*

$DD - S \cup EX \cup e^+$ *consistent* $\forall\ e^+ \in E^+$
$DD - S \cup EX \cup e^-$ *inconsistent* $\forall\ e^- \in E^-$. ❑

From here on we refer to the conjuction of the negated negative examples as *NE, i.e.,* $NE = \bigwedge_{e^- \in E^-} (\neg e^-)$.

Proposition: *Given a CKB-Diagnosis Problem (DD,* E^+, E^-*) , a diagnosis S exists iff*
$\forall e^+ \in E^+$*:* $e^+ \cup NE$ *is consistent.*

Proof. see [5].

Corollary: *S is a diagnosis iff*
$\forall e^+ \in E^+$*:* $DD - S \cup e^+ \cup NE$ *is consistent.* ❑

The following remark relates configuration and diagnosis for configurator knowledge bases [5].

Remark: Let e^+ be partitioned in two disjoint sets e^+_{CONF} and e^+_{SRS} where e^+_{CONF} is a set of positive ground literals whose predicate symbols are in the set of *CONL* and e^+_{SRS}

represents system requirements (if some are specified in conjunction with the positive example).

S is a diagnosis (DD, E^+, E^-) iff $\forall e^+ \in E^+ : e^+_{CONF}$ *is a consistent configuration for* $(NE \cup DD - S, e^+_{SRS}, CONL)$.

Note that if the completeness axioms have been added to e^+_{CONF} then e^+_{CONF} is a valid configuration for $(NE \cup DD - S, e^+_{SRS}, CONL)$.

4 Hierarchies in the Knowledge Base

We will show how hierarchies in the knowledge base can be used for the calculation of diagnoses on the different levels of abstraction. Therefore we assume that the individual constraints from the knowledge base are arranged into named groups which can be again grouped, such that the structure forms a tree. This hierarchical structure *T* can be expressed using a function *sons(n)*, which returns the direct successors of a node *n* in the tree, i.e., the elements of a named group *n* (and ∅ if *n* is a leaf node). We assume a group *root* to exist in the tree representing the root of the tree. The leaf nodes of the tree are the individual sentences from *DD*. A function *leaves(n)* returns all leaf nodes for a given node *n* which are under *n* (and *n* itself if it is already a leaf node). Finally, all diagnosable constraints from *DD* have to be contained in the tree. Note, that the idea for the following framework is that we consider all constraints of a group to be potentially faulty, if at least one constraint of the group is faulty.

Definition (Hierarchy tree): *A hierarchy tree T for a configuration knowledge base DD is a tree, where*

the leaf nodes are named elements from DD,
a node "root" represents the root element of the tree,
inner nodes represent named constraint groups from the knowledge base, and
the names all leaf nodes and inner nodes appear exactly once in the tree. ❑

For hierarchical diagnosis we extend our notion of *CKB-Diagnosis* in a way that also constraint group names can appear in the diagnosis. We define a function *successors(n)* to be returning the set of all direct and indirect successors of a node *n* in the tree (and ∅, if *n* is a leaf node). The function *allLeaves(N)* defined on a set of nodes returns the union of *leaves(n)* applied to every $n \in N$.

Definition (Abstract CKB-Diagnosis): *An Abstract CKB-Diagnosis for a configuration problem (DD,E^+,E^-) and a hierarchy tree T is a set S of nodes of T, such that there exists an extension EX, where EX is a set of logical sentences, such that:*

$DD - allLeaves(S) \cup EX \cup e^+$ *consistent* $\forall e^+ \in E^+$,
$DD - allLeaves(S) \cup EX \cup e^-$ *inconsistent* $\forall e^- \in E^-$. ❑

Definition (Minimal Abstract CKB-Diagnosis): *An Abstract CKB-Diagnosis S for (DD,E^+,E^-) and T is said to be minimal, if no subset S' ⊂ S is an Abstract CKB-Diagnosis.* ❑

In order to ensure that by using this form of abstraction for different levels no diagnostic information is lost, we have to show that every abstract level diagnosis has a corresponding diagnosis at a more detailed level.

Proposition (Soundness of diagnosis): Let *ABSTR* be an Abstract CKB-Diagnosis for a configuration problem (DD,E^+,E^-) and a hierarchy tree *T* then there exists a *CKB*-Diagnosis *DIAG* such that *DIAG* is a subset of *allLeaves(ABSTR)*.

Proposition (Completeness of diagnosis): Let *DIAG* be a *CKB*-Diagnosis for a configuration problem (DD,E^+,E^-) and a hierarchy tree *T* then there exists an Abstract *CKB*-Diagnosis *ABSTR* such that *DIAG* is a subset of *allLeaves(ABSTR)*.

Proof: (see [6]).

5 Computing Diagnoses at Different Levels

Given the above definitions, we can extend the standard hitting-set algorithm for model-based diagnosis to calculate (minimal) diagnoses at the different levels. In the standard algorithm ([11],[16]), *conflict sets* are used for focusing purposes. For the domain of diagnosis of knowledge bases [5], a conflict set is defined as follows:

Definition (Conflict Set): *A conflict set CS for (DD, E^+,E^-) is a set of elements from DD such that*

$\exists e^+ \in E^+$: $CS \cup e^+ \cup NE$ *is inconsistent.* ❑

In order to support calculation of minimal diagnosis at different levels of abstraction, we extend the definition, such that also constraint groups can appear in a conflict set.

Definition (Abstract Conflict Set): *An abstract conflict set for (DD,E^+,E^-) and a hierarchy tree T is a set ACS of elements from T such that*

$\exists e^+ \in E^+$: $allLeaves(ACS) \cup e^+ \cup NE$ *is inconsistent.*❑

For the computation of minimal diagnoses for configurator knowledge bases, the HS-DAG algorithm from ([11], [16]) is adapted as follows: a node *n* in the DAG is labeled by a conflict set *ACS(n)*; edges leading away are labeled by elements $s \in ACS(n)$. The set of edge labels on the path from the root to a node *n* is referred to as *H(n)*. In addition, for each node *n* a set *CE(n)* of consistent positive examples is stored, knowing that once an example is already consistent it will not become inconsistent after further removal of constraints. Since a node can have multiple direct predecessors [11] - referred to as *preds(n)* - we combine the sets *CE* from all direct predecessors for such a node.

According to the idea of iteratively substantiating abstract diagnoses following the hierarchical structure of the problem, we will initially compute a set of high-level diagnoses, which can then be refined to a more detailed level. Consequently, the diagnostic algorithm has an additional input parameter (context) besides the problem description and the examples, i.e., an abstract diagnosis that was already computed on a higher abstraction level. For the calculation of diagnoses on the next level of detail, the constraint groups from the higher-level diagnosis are replaced by their successor nodes according to the hierarchy. Accordingly, given an abstract diagnosis *AD* as

context, the diagnosable components (in terms of model-based diagnosis typically denoted as *COMPS*) for the refined diagnoses are given as follows:

- If $AD = \emptyset$, only elements from *sons(root)* can be contained in the diagnoses.
- If $AD \neq \emptyset$, we have to take a special set of nodes into account for the next refinement step: a) the leaf nodes from *AD*, and b) for each constraint group in *AD*, we have to compute the path of that element to the root of the hierarchy tree. Given the set of nodes that are contained in one of these paths, we have to compute the union of all direct successors of these nodes.

Note, that we have to take these direct successors along the abstraction hierarchy into account for the next-level diagnosis, since additional constraint groups leading to minimal diagnoses can appear in the detail-level diagnoses, which were hidden by the minimality criterion at some abstract level. These special cases of hidden diagnoses are explained in more detail in [6].

Algorithm 1: Diagnosis in abstraction context (schema).
In: (DD,E^+,E^-), *T*, an Abstract Diagnosis *AD*
Out: a set of refined diagnoses *RD*

(1) Use the hitting set algorithm to generate a pruned HS-DAG *D* for the collection *F* of abstract conflict sets for $((DD,E^+,E^-), T, AD)$. Compute the DAG in breadth-first manner in order to generate diagnoses in order of their cardinality.
(a) Every theorem prover call $TP(DD - H(n), E^+ - CE(preds(n)), E^-, T, AD)$ at a node *n* tests whether there exists an $e^+ \in E^+$ such that there is an inconsistency. In this case an (abstract) conflict set is returned, otherwise it returns ok.
(b) Set *CE(n)* to be the set of examples found to be consistent in the call to *TP* union the already consistent examples at the direct precedessors of *n*.
(2) Return {H(n) | n is a node of D labeled by ok}.

6 Computing All Minimal Diagnoses

We propose an iterative approach starting with a high-level diagnosis that can be computed efficiently. The user can decide to stop at this level and focus on some group(s) of constraints or can refine these results to a more concise level. In the following, an algorithm is presented where a tree with nodes labeled with sets of diagnoses is generated, where at each successor node one of the diagnoses of the parent is refined. First, an initial set of top-level diagnoses (in context *root* of hierarchy tree *T*) is generated. Then the tree is generated in breadth-first manner, where for each diagnosis of the parent still containing a constraint group, a child node is generated and diagnosis is performed in the context of that diagnosis. Note that the node is only refined if the considered diagnosis is not already somewhere else in the tree. The algorithm ends, if no more nodes can be refined. Furthermore, if we are only interested in leading diagnoses, the search can be limited, e.g., to a given cardinality. The usage of the standard diagnosis algorithm guarantees that the computed diagnoses are correct and minimal. Furthermore, the result of every refinement step characterizes the candidate space. These candidate spaces include all minimal diagnoses. It follows, that no minimal diagnosis is excluded during refinement.

Algorithm 2: Iterative refinement of diagnoses*(sketch):*
rootnode_diagnoses = diagnose(DD,E⁺,E⁻, T, ∅)
set E⁺ = E⁺ − {e⁺ ∈ E⁺| e⁺ consistent with DD}
*:**label** refine*
refinable = set of diagnoses from current leaf nodes
containing constraint groups.
if** refinable = ∅ **goto** :end; **endif
***forall** d ∈ refinable*
calculate diagnosis d' = diagnose(DD,E⁺,E⁻, T,d)
***if** d' not already in tree*
create child node for d labeled with d'
endif
endfor
***goto** :refine*
*:**label** end*

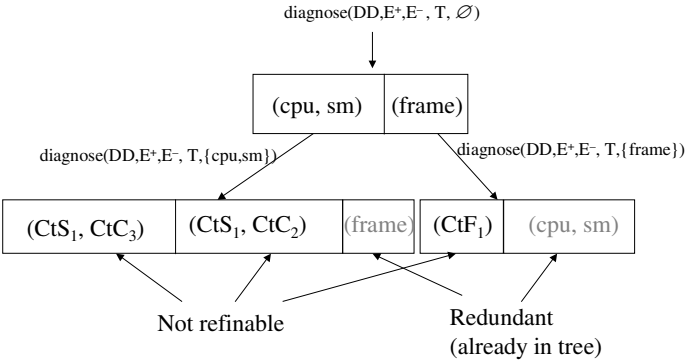


Fig. 1. Tree of diagnoses for example problem

Fig. 1 depicts the outcome of the application of Algorithm 2 to the example problem from Section 2. Fig. 2 shows the hierarchy tree T for the example.

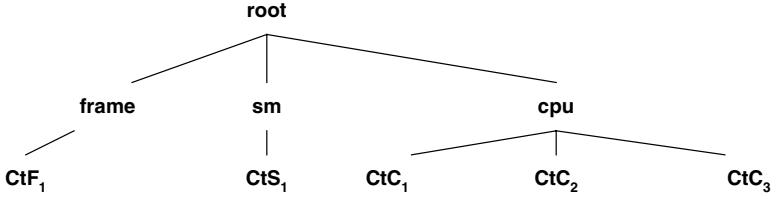


Fig. 2. Hierarchy tree T for example problem

Diagnosis at the top level results in two diagnoses *{ab(sm). ab(cpu).}* and *{ab(frame).}*; these diagnoses are contained in the root node of the tree in Fig. 2. Next, these diagnoses are refined in breadth-first order, i.e., *{ab(sm). ab(cpu).}* is expanded resulting in three diagnoses, namely *{ab(CtS_1). ab(CtC_3).}, { ab(CtS_1). ab(CtC_2).}*, and *{ab(frame)}*. The last one is not added to the refinement tree because it is already contained in the root node of the tree. After that, *{ab(frame).}* is

expanded, resulting in *{ab(CtF$_1$).}* and *{ab(cpu). ab(sm).}*, whereby the latter is again not included in the tree. Next, the algorithm proceeds by trying to refine the diagnoses at the next level; however, in this case, this second level in the tree does not contain any diagnoses that contain refinable components, and the algorithm ends.

7 Implementation

In order to test the applicability of our approaches, we implemented a prototype using the industrial-strength software library *ILOG Configurator* [13]. Using this package of C++ libraries, a configuration problem is formulated in terms of a *Generative Constraint Satisfaction Problem* (GCSP) [7]. This enhancement of the basic CSP mechanism allows the number of variables of the problem to be dynamically changed, and the number of employed components may not be known beforehand. The domain description with the additional constraints and the examples are declared to the configurator using calls to the library. In the context of that CSP, a conflict set is a set of constraints from the knowledge base, which, if canceled, makes the configuration problem satisfiable. Using this library, the search for an arbitrary solution for a configuration problem can be done very efficiently. However, no adequate explanation mechanisms for the calculation of (minimal) conflict sets are provided.

We implemented the diagnostic algorithm both for the flat approach from [5] and the extended hierarchical algorithm. Note, that when using this library, a basic two-level hierarchy, i.e., assignment of constraints to component types is already given with the problem formulation and does not cause additional modeling efforts. The effective computation time for the diagnosis task depends on several factors, e.g., number of constraint types, cardinality of the diagnoses or the time to test one individual example for satisfiability. Diagnosis of the simple example problem can be done nearly instantaneously with both algorithms; the identification of two triple faults in a setting with about twenty types of constraints and about hundred constraint instances is done in a few seconds on a standard PC running Windows NT with both algorithms, whereby our unoptimized prototype does not calculate minimal conflict sets nor utilizes domain dependent heuristics. However, for larger knowledge bases (containing about hundred types of constraints and component types and hundreds of constraint instances), the usage of the hierarchical approach with refinement of the diagnoses leverages the problem of computational complexity. When considering our simple example from Section 2, components will only be considered as one single diagnosable element and will never be expanded to a more detailed level if they are not in any abstract diagnosis. Therefore, even larger and complex knowledge bases remain diagnosable within an acceptable computation time of a few seconds, because additional constraints within the correct parts of the knowledge base only influence the costs of the consistency checks but not those of the diagnostic process. In addition, we conducted preliminary experiments with real-world examples from the domain of private telecommunication systems.

Complexity issues. For a simple analysis of the reduction of the computational complexity when using a hierarchical approach, let us consider the search space for diagnosis candidates for both approaches. Given a set of n diagnosable components, i.e., configuration constraints, where we want to find diagnoses of cardinality k, the

computational complexity is $O(n^k)$. If we are given a two-level hierarchy where the n diagnosable components are distributed over g groups, the number of groups determines the computational complexity $O(g^k)$ for the first level. If we want to refine one of the diagnoses, the constraint groups are replaced with the contained elements from the next level and the average number of elements for two levels will be n/g. Given a number s, $s \leq k \leq g$, describing the number of constraint groups in the abstract diagnosis, the number of diagnosable components for the next level is $(n/g)*s + (g-s)$. The number of remaining constraint groups, which were not in the abstract diagnosis and are therefore not refined, is $(g - s)$. This leads to the complexity of $O((n/g)*s + (g-s))^k)$.

The possible achievable benefits from the hierarchical abstraction depend on several factors: First, the number of needed refinement steps depends on the number of existing diagnoses (the upper bound) and the distribution of elements from the detailed diagnoses among constraint groups. In the best case, all detail-level diagnoses are included in one (or a few) abstract diagnosis, whereas in the worst case all detailed diagnoses correspond to different abstract diagnoses. However, it can be reasonably assumed that only small fractions of the knowledge base are faulty after maintenance. Another factor is how the constraints are grouped, i.e., how many constraints for an individual group exist. In the worst case, each group contains one element leading to additional overhead when using the hierarchies. In good cases, only groups of small size are contained in the abstract diagnoses.

Finally, the assumption that only large, non-minimal conflicts are available following the argumentation of Section 2 (dependencies, expressiveness of configuration language, and specialized inference mechanisms) is an important factor. In cases where minimal conflicts can be easily computed, the hierarchical approach will lead to additional overhead if diagnoses at the detailed levels are needed. Conversely, the more the conflicts contain irrelevant elements, the better the hierarchical approach reduces the complexity.

8 Related Work

Model-based diagnosis techniques were initially developed for the identification of faults in physical devices, e.g., electronic circuits. Later, these techniques were adopted for diagnosis and debugging of software, e.g., logic programs [4], relational database consistency constraints [10], hardware designs specified in VHDL [9], and overconstrained Constraint Satisfaction Problems [2]. Our work extends the work of [5] by exploiting hierarchies for consistency-based diagnosis of configuration knowledge bases. The usage of hierarchies for the diagnosis task has been discussed in various application areas of model-based diagnosis (e.g., [8],[12],[15],[17]). Our approach mostly corresponds to what is called *structural abstraction* (vs. *behavioral abstraction*) and aims at a more efficient diagnosis process. One of the important problems is to have the information on the hierarchy available at each abstraction level (causing additional modeling effort). For the case of debugging of configuration knowledge bases, however, the hierarchical abstraction has a good correspondence to the configurable artifact. Changes to the product catalog are usually applied to sets of

modules (configuration components) leading to a small set of effectively affected components.

In [1], it was shown that when modeling a system at different levels of abstraction (independently) for general diagnosis problems there may be situations where diagnoses at a detailed level do not have a correspondence to a diagnosis on a more abstract level such that diagnostic information may be lost. This phenomenon cannot appear in our approach, because at each level, the system's "behavior" (consistency checks) is always analyzed on the most detailed level.

[12] describes hierarchical diagnosis based on value propagation and with XDE an extension of the ATMS approach. Our approach is similar in the way structural decomposition is applied. However, our goal was to integrate configuration engines (e.g., based on generative constraint satisfaction) and diagnosis. The approach of [16] offers an appealing way for this integration, which was extend by our work in order to employ hierarchies. [15] also uses hierarchies for improving the efficiency of diagnosis but applies a different notion of diagnosis by defining a diagnosis as a logical consequence of a theory.

Different approaches to diagnosis which avoid the computation of conflict sets were proposed by [3] and [18]. They improve the underlying theorem proving algorithms such that diagnoses can be computed efficiently. Note, that our goal was to reuse specialized problem solvers, which are optimized to solve complex configuration problems and not to provide conflict sets of explanations. The incorporation of diagnosis techniques from [3] and [18] without degrading the performance of the configurators remains an interesting open issue.

9 Conclusions

The demand for (AI-based) product configuration technology is steadily increasing. For validation and maintenance tasks only limited support can be found in nowadays systems. For these tasks it was shown in [5] how techniques from model-based diagnosis can support the knowledge engineer in validating the knowledge base.

Currently, for the configuration of large and complex products, special problem solving mechanisms must be applied. In addition, the domain requires general clauses for expressing configuration constraints. Due to these facts, only limited explanation of conflicts is provided. We showed how the exploitation of hierarchical structures can significantly improve the efficiency of diagnosing configuration knowledge bases. This was achieved by employing specialized configuration systems for consistency checking and extending the approach from [16] in order to cope with hierarchies. We have presented a sound and complete algorithm relying on iterative refinement of diagnoses and validated our approach in a prototype implementation.

References

1. K. Autio and R. Reiter. Structural abstraction in model-based diagnosis, Proc: ECAI'98, Brighton, UK, John Wiley and Sons, 1998, pp. 269-273.

2. R.R. Bakker and F. Dikker and F. Tempelman and P.M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. Proc: IJCAI'93, Chambery, Morgan Kaufmann, 1993, pp. 276-281.
3. P. Baumgartner, P. Fröhlich, U. Furbach and W. Nejdl. Semantically Guided Theorem Proving for Diagnosis Applications. Proc: IJCAI'97, Nagoya, Morgan Kaufmann, 1997, pp. 460-465.
4. L. Console, G. Friedrich, and D.T. Dupré. Model-based diagnosis meets error diagnosis in logic programs. Proc: IJCAI'93, Chambery, Morgan Kaufmann, 1993, pp. 1494-1501.
5. A. Felfernig, G.Friedrich, D. Jannach, and M. Stumptner. Consistency based diagnosis of configuration knowledge bases. Proc: ECAI'2000, Berlin, IOS Press, 2000, pp. 146-150.
6. D. Jannach. Integration of consistency-based diagnosis and configuration, PhD thesis, University Klagenfurt, 2001.
7. G. Fleischanderl, G. Friedrich, A. Haselboeck, H. Schreiner and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction, IEEE Intelligent Systems, July/August, 1998.
8. G. Friedrich, Theory Diagnosis. A Concise Characterization of Faulty Systems, Proc: IJCAI'93, Chambery, France, Morgan Kaufmann, 1993, pp. 1466-1473.
9. G. Friedrich, M. Stumptner, and F. Wotawa. Model-Based Diagnosis of Hardware Designs, Artificial Intelligence (111) 2, Elsevier, 1999, pp. 3-39.
10. M. Gertz, U. Lipeck. A Diagnostic Approach to Repairing Constraint Violations in Databases. Proc: DX'95 Workshop, Goslar, 1995.
11. R. Greiner, B.A. Smith, R.W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. Artificial Intelligence, 41(1), Elsevier, 1989, pp. 79-88.
12. W. Harmscher. Modeling Digital Circuits for Troubleshooting, Artificial Intelligence 51(1-3), Elsevier, 1991, pp. 223-271.
13. D. Mailharro. A Classification and Constraint-based Framework for Configuration, AI EDAM, Vol 12(98), Cambridge University Press, 1998.
14. S. Mittal and F. Frayman. Towards a generic model of configuration tasks, Proc: IJCAI'89, 1989, pp. 1395-1401.
15. I. Mozetic. Hierarchical Model-based Diagnosis, in: W. Harmscher et al.: Readings in Model-based Diagnosis, Morgan Kaufmann, 1992, pp. 354-372.
16. R. Reiter. A theory of diagnosis from first principles. Artificial Intelligence, 32(1), Elsevier, 1987, pp. 57-95.
17. P. Struss. What's in SD? Towards a theory of Modeling of Diagnosis. In: W. Harmscher et al.: Readings in Model-based Diagnosis, Morgan Kaufmann, 1992.
18. M. Stumptner, F. Wotawa. Diagnosing tree-structured systems, Proc: IJCAI'97, Nagoya, Morgan Kaufmann, 1997, pp. 440-445.

Towards Distributed Configuration

Alexander Felfernig, Gerhard E. Friedrich, Dietmar Jannach, and Markus Zanker

Institut f. Wirtschaftsinformatik und Anwendungssysteme
Universität Klagenfurt
email{felfernig, friedrich, jannach, zanker}@ifit.uni-klu.ac.at

Abstract. Shorter product cycles, lower prices, and the production of highly variant products tailored to the customer needs are the main reasons for the proceeding success of product configuration systems. However, today's product configuration systems are designed for solving local configuration tasks only, although the economic development towards webs of highly specialized solution providers demands for distributed problem solving functionality. In this paper we motivate the integration of several configurators and give a formal definition of the distributed configuration task based on a logic theory of configuration. Furthermore, we present a basic architecture comprising several configuration agents and propose an algorithm for cooperation between distributed configuration systems that ensures correctness and completeness of configuration results.

1 Introduction

Configurators are not only important enablers of the mass customization paradigm but also among the most successful applications of AI-technology. Configurators calculate product variants which fulfill customer requirements as well as technical and non-technical constraints on the product solution. As the digital economy of the 21^{st} century will be based on flexibly integrated webs of highly specialized solution providers, the joint configuration of organizationally and geographically distributed products and services must be supported. The rush for supply chain integration by web-based selling systems and electronic procurement offers new challenges for configuration technology. While supply chain integration of standardized, mostly well defined products can be quite well achieved, the case for complex configurable products and services is still an open research issue.

Current configuration technology [11] does not yet offer concepts and tools to support the integration of configuration systems. In particular, a distributed configuration problem cannot be solved by a single configurator with a centralized knowledge base for security and privacy reasons of suppliers. As we have to cope with distributedness, we must accept the fact that there is no central point of knowledge. Neither a main vendor nor any of its suppliers has full knowledge on the whole problem domain. The second point is heterogeneity; we cannot assume that each of the engaged parties, that have knowledge on a subset of the problem domain and can therefor provide to the overall solution, employs the same knowledge representation formalisms.

Consequently, our goal is to contribute to the further development of configuration technology such that distributed configuration problems can be solved. In order to give a

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 198–212, 2001.

clear and general problem definition, we base our contribution on a logical foundation. This abstract approach allows us to determine the basic requirements for the integration of special instances of configuration methods and tools found in research and industry. Our introductory example is based on an application scenario provided by one of our industry partners (Section 2). Based on the definition of a central configuration problem, we formally define the distributed configuration task (Section 3) and show under which conditions the central and distributed problems are equivalent. We present an algorithm which enables configuration agents to cooperatively construct valid configurations and describe the required properties in order to assure correctness and completeness (Section 4). Aspects of integration are adressed in Section 5. Finally we discuss related work followed by conclusions.

2 Motivating Example

We introduce our concepts by presenting a motivating scenario from the area of telecommunication systems. Our product example is a telecommunication switch for enterprise networks. The functionality of the switch can be extended by installing additional software modules onto the hardware component such as management software or application packages for messaging and ip-services. These additional applications are third-party products or may be developed by a subsidiary company. The customer, however, wishes to order a completely configured product solution, comprising the switching hardware and all needed add-ons. For obvious reasons each supplier maintains the product knowledge within its own sales configuration system that cooperates with others. In our scenario a facilitating agent coordinates the search for a configuration solution of three configuration agents representing the providers of the *switching hardware*, the *messaging* and the *ipvoice* application software add-ons. We employ a logic theory of configuration [2] that complies with the component-port representation for configuration knowledge [10]. This logical model serves as a general ontology for the configuration domain. We allow that all involved configurators may use a proprietary representation formalism. However it must be assured that the content of communicated messages can be mapped onto the concepts of this logical theory. In our example we use only $types$ to denote the set of component types while $ports$ describes their connection points. We do not employ attributes of types and their domains in this example, although these concepts are part of our logic theory of configuration.

$types = \{tecom, srack, lrack, ipvoice, swpack1, swpack2, msger, uppack\}.$
$ports(tecom) = \{rack, ipvoice, msger\}.$
$ports(lrack) = \{tecom\}.\ ports(srack) = \{tecom\}.$
$ports(ipvoice) = \{tecom, swpack\}.$
$ports(swpack1) = \{ipvoice\}.$
$ports(swpack2) = \{ipvoice\}.$
$ports(msger) = \{tecom, upgr\}.$
$ports(uppack) = \{msger\}.$

The predicates used for describing configurations are contained in a set $CONL$, where $CONL = \{type/2, conn/4\}$ for our example. A type t is associated with a component c by literal $type(c, t)$. A connection is represented by literal $conn\ (c1, p1, c2, p2)$ where $p1$ (resp. $p2$) is a port of component $c1$ (resp. $c2$). Usually an attribute value v assigned to attribute a of component c is represented by a literal $val(c, a, v)$. In our example, we omit val - predicates to keep the presentation short. The configuration knowledge of each of the three involved configurators is defined by a domain description (DD) comprising sets of logical sentences that specify compatibility constraints and the derivation of additional facts. In addition to the constraints C_i listed below, a set of application independent sentences denoted by C_{basic} is included in the domain description, specifying that connections are symmetric, that a port can only be connected to one other port, and that components have a unique type.

$DD_{switch} = \{C_1,\ C_2\} \cup C_{basic}$.
$DD_{ip} = \{C_3,\ C_4\} \cup C_{basic}$. $DD_{msg} = \{C_5\} \cup C_{basic}$.

C_1 : *"If the switch has more than 200 end devices then a large rack is needed."*

$\forall T, C : type(T, tecom) \wedge devices(T, C) \wedge$
$C > 200 \Rightarrow \exists L :$
$type(L, lrack) \wedge conn(L, tecom, T, rack)$.

C_2 : *"If the customer requires voice-over-ip then the ipvoice application must be installed."*

$\forall T : type(T, tecom) \wedge voice\text{-}over\text{-}ip(T) \Rightarrow \exists I :$
$type(I, ipvoice) \wedge conn(I, tecom, T, ipvoice)$.

C_3 : *"An ipvoice application consists either of a swpack1 or swpack2 software module."*

$\forall I : type(I, ipvoice) \Rightarrow \exists P :$
$(type(P, swpack1) \vee type(P, swpack2)) \wedge$
$conn(P, ipvoice, I, swpack)$.

C_4 : *"A swpack1 software module is incompatible with upgrade uppack."*

$\forall T, I, M, P1, U : type(T, tecom) \wedge type(I, ipvoice) \wedge$
$conn(I, tecom, T, ipvoice) \wedge type(M, msger) \wedge$
$conn(M, tecom, T, msger) \wedge type(P1, swpack1) \wedge$
$conn(P1, ipvoice, I, swpack) \wedge type(U, uppack) \wedge$
$conn(U, msger, M, upgr) \Rightarrow false$.

C_5 : *"If the software msger is sold together with the ipvoice application then it must contain the upgrade uppack."*

$\forall T, M, I : type(T, tecom) \wedge type(M, msger) \wedge$
$conn(M, tecom, T, msger) \wedge type(I, ipvoice) \wedge$
$conn(I, tecom, T, ipvoice) \Rightarrow \exists P :$
$type(P, uppack) \wedge conn(P, msger, M, upgr)$.

In our domain a system requirements specification (SRS) provided by the customer is only sent to the switching hardware manufacturer. It is a logic theory that comprises predicates from $CONL$ as well as any other predicates that specify the requirements a customer wants to be fulfilled.

$$SRS_{switch} = \{\exists T, M : type(T, tecom) \wedge \\ devices(T, 300) \wedge voice\text{-}over\text{-}ip(T) \wedge \\ type(M, msger) \wedge conn(M, tecom, T, msger).\}$$

Given the above constraints and customer requirements, central problem solving would achieve the following complete and consistent configuration result[1]:

$$CONF = \{type(id_1, tecom).\ type(id_2, lrack). \\ type(id_3, ipvoice).\ type(id_4, msger). \\ type(id_5, swpack2).\ type(id_6, uppack).\}$$

Note however, that a central approach is not feasible for security and privacy concerns of involved business entities and the question is how to solve this task for the distributed case. Therefor, we aim at defining the distributed configuration problem and at stating the conditions under which the distributed solving generates equivalent solutions to a central approach.

3 Formalizing Distributed Configuration

In the general framework of [2], a configurator knowledge base consists of a set of logical sentences DD describing available component types, their attributes and connection points as well as constraints on legal product constellations. As sketched in Section 2, configuration problems are solved according to a system requirements specification SRS and the configuration result can be described by means of a set of positive ground literals using predicate symbols from $CONL$.

3.1 Central Configuration Approach

Definition (Configuration problem): *A configuration problem is described by a triple $(DD, SRS, CONL)$, where DD and SRS are sets of logical sentences and $CONL$ is a set of predicate symbols. DD represents the domain description, SRS the system requirements specification for a configuration problem instance. A configuration $CONF$ is described by a set of positive ground literals[2] whose predicate symbols are in $CONL$.*□

[1] For reasons of presentation, we employ only $type/2$ predicates for representing configurations and omit the $conn/4$ predicates

[2] By using Skolem constants we have decoupled the representation of a configuration solution from the problem description. Therefor validity of configurations is independent of a bijective renaming of these constants.

Definition (Consistent configuration): *Given a configuration problem* $(DD, SRS, CONL)$*, a configuration* $CONF$ *is consistent iff* $DD \cup SRS \cup CONF$ *is satisfiable.* □

To ensure the completeness of a configuration, additional formulae for each symbol in $CONL$ have to be introduced to $CONF$, e.g., for the $type$ predicate:

$$type(X, Y) \Rightarrow type(X, Y) \in CONF.$$

We denote the configuration $CONF$ extended by these axioms with $\widehat{CONF}$.

Definition (Valid and irreducible configuration): *Let* $(DD, SRS, CONL)$ *be a configuration problem. A configuration* $CONF$ *is valid iff* $DD \cup SRS \cup \widehat{CONF}$ *is satisfiable.* $CONF$ *is irreducible if there exists no other valid configuration* $CONF^{sub}$ *such that* $CONF^{sub} \subset CONF$. □

3.2 Distributed Configuration Approach

Definition (Distributed configuration problem): *A distributed configuration problem for n different configuration agents is described by a triple* $(DD_{set}, SRS_{set}, CONL)$ *where*

$DD_{set} = \{DD_1, \ldots, DD_n\}$ *and*
$SRS_{set} = \{SRS_1, \ldots, SRS_n\}$.

Each element of DD_{set} *and of* SRS_{set} *is a set of logical sentences and* $CONL$ *is a set of predicate symbols. For* $k \in \{1, \ldots, n\}$, DD_k *corresponds to the domain description of the configuration system* k *and* SRS_k *specifies its system requirements. A configuration* $CONF$ *is described by a set of positive ground literals whose predicate symbols are in* $CONL$.□

Remark: In extension to the introductory example, all configuration agents can be initialized with an individual system requirements specification contained in the set SRS_{set}.

Definition (Valid solution to a distributed configuration problem): *Given a distributed configuration problem* $(DD_{set}, SRS_{set}, CONL)$*, a configuration* $CONF$ *is valid iff* $DD_k \cup SRS_k \cup \widehat{CONF}$ *is satisfiable* $\forall k \in \{1, \ldots, n\}$.□

In practice configurators of suppliers collaborate by exchanging (partial) configurations, i.e., these configurators can be seen as independent modules jointly constructing a common solution. Related to our case this implies that the domain descriptions $DD_1 \ldots DD_n$ and the system requirements $SRS_1 \ldots SRS_n$ of each configurator are independent except assertions regarding the configuration. We achieve this property by using disjoint sets of predicate symbols in each DD_k and SRS_k but allow the joint use of predicate symbols contained in $CONL$, i.e., for every pair of configurators i, j

$(psymbols^3(DD_i) \cup psymbols(SRS_i)) \cap (psymbols(DD_j) \cup psymbols(SRS_j)) = X$, where $X \subseteq CONL$. This way, dependencies, that surpass the knowledge of local companies, are considered via their effects on the configuration result. We call this property *defined interfacing*.

Theorem: *Let* $(DD, SRS, CONL)$ *be a configuration problem and* $(DD_{set}, SRS_{set}, CONL)$ *a distributed configuration problem with defined interfacing where*

$DD = \bigcup_{dd \in DD_{set}} dd$ *and*

$SRS = \bigcup_{srs \in SRS_{set}} srs$.

$CONF$ *is a valid configuration for* $(DD, SRS, CONL)$ *iff* $CONF$ *is a valid solution for the distributed configuration problem* $(DD_{set}, SRS_{set}, CONL)$.□

Proof (sketch):

($\Rightarrow$) Since $DD \cup SRS \cup \widehat{CONF}$ is satisfiable, and $DD_k \subseteq DD$, $SRS_k \subseteq SRS$ also $DD_k \cup SRS_k \cup \widehat{CONF}$ is satisfiable. It follows $CONF$ is also a valid solution for the distributed configuration problem $(DD_{set}, SRS_{set}, CONL)$. □

($\Leftarrow$) $DD_k \cup SRS_k \cup \widehat{CONF}$ (we call this theory T_k) and $DD_j \cup SRS_j \cup \widehat{CONF}$ (called T_j) with $k \not= j$ are consistent. Let us assume $DD_k \cup SRS_k \cup \widehat{CONF} \cup DD_j \cup SRS_j$ is inconsistent. It follows that $T_k \vdash \neg(DD_j \cup SRS_j)$. The theory $(DD_j \cup SRS_j)$ can be transformed to an equivalent theory expressed by a set of clauses Cs_j. Consequently, T_k has to imply the negation of a clause C_{CONL} where C_{CONL} follows from Cs_j. Note, that T_k can only imply such a clause $\neg C_{CONL}$ which solely consists of predicates of CONL since T_k and T_j have only predicates in common which are in CONL. Because $\widehat{CONF} \subseteq T_k$ is a complete theory w.r.t. predicates in CONL it follows that $\widehat{CONF} \vdash \neg C_{CONL}$. However, $DD_j \cup SRS_j$ implies C_{CONL} and therefore T_j is inconsistent which is a contradiction to the fact that T_j is consistent. Consequently, $DD_k \cup SRS_k \cup \widehat{CONF} \cup DD_j \cup SRS_j$ is consistent. By applying this argument to all elements of DD_{set} and SRS_{set} it follows that $\bigcup_{dd \in DD_{set}} dd \bigcup_{srs \in SRS_{set}} srs \cup \widehat{CONF}$ is consistent. □

Corollary: *Let* $(DD, SRS, CONL)$ *be a configuration problem and* $(DD_{set}, SRS_{set}, CONL)$ *a distributed configuration problem with defined interfacing where*

$DD = \bigcup_{dd \in DD_{set}} dd$ *and*

$SRS = \bigcup_{srs \in SRS_{set}} srs$.

A valid configuration $CONF$ *for* $(DD, SRS, CONL)$ *is irreducible iff* $CONF$ *is a valid solution to the distributed configuration problem* $(DD_{set}, SRS_{set}, CONL)$ *and there exists no other valid solution* $CONF^{sub}$ *to the distributed configuration problem such that* $CONF^{sub} \subset CONF$. □

[3] The function $psymbols(T)$ returns all predicate symbols that are employed in the logical theory T.

3.3 Conflicts

When solving a configuration problem, partial solutions are extended with the goal to generate valid configurations. During the problem solving phase it could be discovered that such partial solutions are in conflict with $DD \cup SRS$. As a consequence these conflicts must guide the subsequent search process in order to avoid the rediscovery of inconsistent configurations. Due to the *defined interfacing* property, conflicts can only be caused by predicate symbols from $CONL$ in $CONF$, i.e.:

Definition (Conflict): *Let* $(DD, SRS, CONL)$ *be a configuration problem and* $CONF$ *be a consistent set of sentences in* $CONL$. $CONF$ *is a conflict of* $(DD, SRS, CONL)$ *iff* $SRS \cup DD \vdash \neg CONF$. □

The relation between conflicts and configurations is described as follows.

Theorem: *Let* $(DD, SRS, CONL)$ *be a configuration problem,* $NG-CONFLICTS$ *is its set of negated conflicts and* $\widehat{CONF}$ *be a configuration including the completeness axioms.* $\widehat{CONF}$ *is a valid configuration iff* $\widehat{CONF} \cup NG - CONFLICTS$ *is satisfiable.* □

Proof (sketch):

($\Rightarrow$) Since $DD \cup SRS \cup \widehat{CONF}$ is satisfiable and $NG - CONFLICTS$ is entailed by $DD \cup SRS$ it follows that $\widehat{CONF} \cup NG - CONFLICTS$ is satisfiable. □

($\Leftarrow$) Let $\widehat{CONF}$ be a configuration where $\widehat{CONF} \cup CONFLICTS$ is satisfiable. Suppose $\widehat{CONF}$ is not a valid configuration i.e. $DD \cup SRS \cup \widehat{CONF}$ is unsatisfiable. Therefore, $DD \cup SRS \vdash \neg\widehat{CONF}$. But then $\widehat{CONF}$ would be a conflict and $\neg\widehat{CONF}$ must be included in $NG-CONFLICTS$, contradicting the fact that $\widehat{CONF} \cup NG-CONFLICTS$ is satisfiable. □

Definition (Minimal conflict): *Let* $(DD, SRS, CONL)$ *be a configuration problem and CONF a conflict. CONF is a minimal conflict of* $(DD, SRS, CONL)$ *iff for all conflicts* $CONF^{sub}$ *either* $\neg CONF^{sub} \not\vdash CONF$ *or* $\neg CONF^{sub} \equiv \neg CONF$. □

Note, that when searching for valid configurations we must achieve consistency with the negated conflicts. Since the negation of minimal conflicts implies the negation of non-minimal conflicts, thus a non-minimal conflict needs not to be considered. The same argument holds for an equivalent conflict. In the following we relate conflicts of the central and the distributed configuration approach.

Corollary: *Let* $(DD, SRS, CONL)$ *be a configuration problem,* $\widehat{CONF}$ *be a configuration including the completeness axioms, and* $(DD_{set}, SRS_{set}, CONL)$ *a distributed configuration problem with defined interfacing where*

$DD = \bigcup_{dd \in DD_{set}} dd$ *and*
$SRS = \bigcup_{srs \in SRS_{set}} srs$.

$\widehat{CONF}$ *is a conflict for* $(DD, SRS, CONL)$ *iff there exists a* $k \in \{1, \ldots, n\}$ *s.t.* $\widehat{CONF}$ *is a conflict for* $(DD_k, SRS_k, CONL)$. □

Note, that every conflict found by a local configuration agent is a conflict for the complete (central) configuration problem. These conflicts must be communicated among the agents, in order to ensure that superfluous work on conflicting configurations is avoided.

4 Basic Model for Interaction

In order to show the feasibility of distributed configuration problem solving according to the above definitions we outline an architectural setting of cooperating agents and propose an algorithm for interaction. Note however, that this model represents a theoretical framework for the general case. For our concrete implementation in an application-oriented international research project we exploit domain specifics of configuration problem solving as described in the next subsection *Extensions for Efficiency*. The configuration knowledge is distributed over a set of *n* configuration agents which may configure concurrently. The communication among them is coordinated via a facilitator agent that collects the (partial) configurations from each agent and distributes them among the others. So there is no direct communication between configurators, but only indirect via the facilitator. This architecture is chosen because it poses less requirements on the capabilities of configuration agents than *peer-to-peer* communication, where each agent must be able to distinguish between several communication channels. As soon as a configuration agent detects a conflict with the joint configuration, the others are informed and measures for conflict resolution are taken. This resolution strategy ensures that a conflict never occurs twice during a session and that the non-existence of a valid configuration for the overall task is detected. We do propose a very general negotiation strategy for conflict resolution, because we consider the integration of already existing configuration systems into this framework.
The configurator agents communicate only with the faciliator, where the exchanged messages have the following signatures:

- $request_k^{no}(CONF^{s_i})$: The configurator *k* receives the configuration $CONF^{s_i}$ and checks if is locally satisfiable. s_i denotes the search depth of the algorithm for this intermediate configuration solution and no counts the interaction cycles.
- $reply_k^{no}(CONF_k^{s_{i+1}})$: The configurator *k* communicates the configuration result $CONF_k^{s_{i+1}}$ in reply to $request_k^{no}(CONF^{s_i})$ back to the facilitator. $CONF_k^{s_{i+1}}$ is a valid local configuration of configurator *k*.
- $conflict_k(CONF^{s_j})$: With this message the configurator *k* alerts the facilitator, that $CONF^{s_j}$ is not satisfiable with its local knowledge base.
- $add\text{-}conflict(C)$: Once the facilitator was alerted with a $conflict$ message, it broadcasts this conflict C to all configuration agents that is then negated and added to their local system requirements SRS_k.

The **facilitator agent** initially distributes only the non-empty sets of individual system requirements SRS_k to the configuration agents that store them - see Algorithm (a). Obviously, requesting a configuration from a configurator without any system requirements would produce merely incidental results and is therefor avoided. Then the facilitator starts the problem solving process by broadcasting $request_k^1(\{\})$ to each recipient of a non-empty SRS_k and awaits $reply_k^1(CONF_k^{s_1})$ messages from these

configuration agents, only (b.2). After collection of replies (b.2) the facilitator unifies the locally completed configurations and broadcasts them to all configuration agents with a $request_k^{no}(\bigcup_k CONF_k^{s_i})$ message. This is now possible because the intermediate result $CONF^{s_i}$ restricts the further solution search of agents. In case at least one of the remote configurators replies a $conflict_k(CONF^{s_j})$ message the facilitator initiates the conflict resolution strategy (c). Here we chose a strategy where it is in the responsibility of every single agent of not delivering a conflicting configuration twice. This is achieved because the facilitator communicates the conflicting configuration $CONF^{s_j}$ to all agents via an $add\text{-}conflict(CONF^{s_j})$ message and backtracks by demanding another reply to $request_k^{no}(CONF^{s_{j-1}})$ (c.1). All replies to the previous request are discarded (b.1). The algorithm terminates either with a valid solution or detects that there exists no solution (c.2). The latter case is implied if the empty set is not satisfiable with the local knowledge base including stored conflicts. A valid configuration for the overall configuration task is found, when no configurator has added new ground facts to $CONF^{s_i}$ during an interaction cycle (b.4), i.e., $CONF^{s_i} = CONF^{s_{i+1}}$.
When a **configuration agent** receives a $request_k^{no}(CONF^{s_i})$ message the algorithm distinguishes between two problem solving levels (d).

First the satisfiability of the received configuration is tested.

Definition (Local satisfiability): *Local satisfiability for agent k is given, iff configuration $CONF^{s_i}$ is satisfiable with its local knowledge base: $DD_k \cup SRS_k \cup CONF^{s_i}$ is satisfiable.*

If *Local Satisfiability* is given, the configuration agent completes the initial configuration $CONF^{s_i}$ to a locally valid configuration $CONF_k^{s_{i+1}}$ (d.1), which is performed in the algorithm by the function $configure$.

Definition (Local validity): *Local validity for agent k is given, iff configuration $CONF_k^{s_{i+1}}$ is a valid configuration w.r.t. its local knowledge base: $CONF_k^{s_{i+1}}$ is valid for $(DD_k, SRS_k, CONL)$.*

If *Local Satisfiability* is not given, the configuration agent replies with a $conflict_k$ $(CONF^{s_i})$ message to the initial request $request_k^{no}(CONF^{s_i})$ (d.2).

When a configurator receives a $add\text{-}conflict(C)$ message it stores it by expanding its local system requirements (e):

$$SRS_k^{'} = SRS_k \cup \neg C.$$

At this stage redundant clauses (e.g., non-minimal conflicts) can be removed from SRS_k.

Algorithm - *Behaviour of facilitator agent*

(a) **initialize**(SRS_{set}) **do**
$\forall SRS_k \not= \{\} : \ forward\ SRS_k$ *to agent k;*
$CONF^{s_0} = \{\}$; *count* $= 1$;
$\forall SRS_k \not= \{\} : \ send(request_k^1(CONF^{s_0}))$;
end do;

(b) **when received**$(reply_k^{no}(CONF_k^{s_i}))$ **do**
(b.1) if $no = count$ *then*
$CONF^{s_i} = CONF_k^{s_i} \cup CONF^{s_i}$;
(b.2) if $(\forall k\{\in 1, \ldots, n\} : \ received(CONF_k^{s_i})) \vee$
$((count = 1) \wedge$
$(\forall SRS_k \not= \{\} : \ received(CONF_k^{s_i})))$ *then*
(b.3) if $CONF^{s_i} \not= CONF^{s_{i-1}}$ *then*
count++;
$\forall k \in \{1, \ldots, n\} : \ send(request_k^{count}(CONF^{s_i}))$;
(b.4) else
terminate algorithm, out: $CONF = CONF^{s_i}$;
end if
end if
end if
end do;

(c) **when received**$(conflict_k(CONF^{s_j}))$ **do**
(c.1) if $CONF^{s_j} \not= \{\}$ *then*
$\forall k \in \{1, \ldots, n\} : \ send(add\text{-}conflict(CONF^{s_j}))$;
count++;
$\forall k \in \{1, \ldots, n\} : \ send(request_k^{count}(CONF^{s_{j-1}}))$;
(c.2) else
terminate algorithm, out: no configuration exists;
end if
end do;

Algorithm - *Behaviour of configuration agent*

(d) **when received**$(request_k^{no}(CONF^{s_i}))$ **do**
(d.1) if $locally\ satisfiable(DD_k \cup SRS_k \cup CONF^{s_i})$ *then*
$CONF_k^{s_{i+1}} = configure(CONF^{s_i})$;
$send(reply_k^{no}(CONF_k^{s_{i+1}}))$;
(d.2) else
$send(conflict_k(CONF^{s_i}))$;
end if
end do;

(e) **when received(** $add\text{-}conflict_k(C)$ **) do**
$SRS_k = SRS_k \cup \neg C$;
end do;

4.1 Extensions for Efficiency

Configuration problems have the property to be usually underconstrained and there exist many good solutions that can be accepted from the standpoint of a domain expert. Furthermore, similar to a centralized approach, heuristics exist to guide the solution search within and between the configuration agents. These allow us to avoid an inefficient *blind* search of the solution space and provide optimized solutions according to some criteria such as price or quality. For design of our prototype implementation we identified the following approaches:

- In a realistic economic setting one destined configuration agent will act as a main vendor that also fulfills the task of the facilitator. Further some partial sequentialization between configurators can be assumed, i.e., the main manufacturer or service provider will configure locally as far as possible and restrict this way the solution space of its suppliers. When reducing the degree of parallelism of solution search the probability for conflict occurence can be obviously diminished. Restricting concurrent solution search to agents whose configuration results do not have side-effects on each other is an additional heuristic. Further a partial ordering of configurators can be used for an advanced negotiation strategy for conflict resolution, where agents with lower priority are the first ones to repair their configuration results. Such a scenario where all configuration agents sequentially add new predicates to the calculated configuration of the predecessor in a supply chain is a specific instantiation of the more general model presented in this paper.
- When configuring complex telecommunication systems computed configurations tend to become quite large with $CONF$ encompassing thousands of facts. It is obvious that not all components and connections of the switching node are of interest to the configuration agent that determines the configuration of the add-on product. Therefor measures towards intelligent filtering of the message content need to be taken. We can assume that configuration knowledge is not randomly partitioned. Based on this partitioning of configuration knowledge, we are able to identify the vocabulary of product domain concepts that specifies the configuration capabilities and informational interest on components and connections for which it has constraints defined in its local knowledge base. By employing domain ontologies we can give an abstract description of the product domain of the configuration agent. Therefor only those facts of the overall configuration solution need to be communicated to a specific configuration agent that correspond to its domain ontology.
- A further step towards lower space complexity is the reduction of conflict size. This can be achieved, if configuration agents are capable of generating minimal conflicts following the definition in Section 3.3. Techniques from model based diagnosis can be exploited to improve conflict generation.

4.2 Solving the Example

In the following we show how to solve the example from Section 2 with our algorithm. In the example three agents are involved, i.e., $k \in \{switch, ip, msg\}$. The problem solving phase starts when the facilitator agent forwards the system requirements SRS_{switch}

to the switching hardware manufacturer and initiates the solution search by sending $request^1_{switch}(\{\})$. Therefor the agent *switch* is the only one to reply to the facilitator in the first cycle of interaction:

(1) $reply^1_{switch}(\{type(id_{s1}, tecom).\ type(id_{s2}, lrack).$
$type(id_{s3}, ipvoice).\ type(id_{s4}, msger).\})$

The facilitator distributes the received configuration as $CONF^{s_1}$ to all agents.

(2) $reply^2_{switch}(CONF^{s_1} \cup \{\})$
$reply^2_{ip}(CONF^{s_1} \cup \{type(id_{i1}, swpack1).\})$
$reply^2_{msg}(CONF^{s_1} \cup \{type(id_{m1}, uppack).\})$

The facilitator unifies the received partial configurations and broadcasts a $request^3_k$ $(CONF^{s_2})$ to all agents.

(3) $conflict_{ip}(CONF^{s_2})$ *because of Constraint* C_4.

The facilitator discards the $reply^2_k$ messages from agents *switch* and *msg*. It broadcasts $add\text{-}conflict(CONF^{s_2})$ and afterwards backtracks with $request^4_k(CONF^{s_1})$ to all agents.

(4) $reply^4_{switch}(CONF^{s_1} \cup \{\})$
$reply^4_{ip}(CONF^{s_1} \cup \{type(id_{i2}, swpack2).\})$
$reply^4_{msg}(CONF^{s_1} \cup \{type(id_{m2}, uppack).\})$

The facilitator generates the union set of all received partial configurations and broadcasts them for another cycle of interaction. Now, no one detects a conflict. All configuration agents determine the validity of the configuration and do not need to derive additional facts. Therefor the algorithm terminates with the same solution as with central problem solving.

4.3 Analysis

For analysis we employ the basic assumption that all configuration agents are capable of generating valid configuration results and are complete w.r.t. the set of all valid configurations, which we assume to be limited for practical reasons. This can be achieved by limiting the number of possible components in an artifact. In order to show the *soundness* of the algorithm we must show that each generated solution $CONF$ satisfies the criteria of a valid distributed configuration stated in section 3. This is given, because $\forall k \in \{1, \ldots, n\} : DD_k \cup SRS_k \cup \widehat{CONF}^{s_i}$ is satisfiable and $CONF^{s_i} = CONF$.
For proofing the *completeness* of the algorithm we must show that if a solution exists the algorithm terminates with a configuration solution $CONF$, otherwise the algorithm terminates with a failure indication. Let us first assume that the algorithm terminates. This happens either by giving a correct solution, or terminating, because no solution exists: $\not\exists CONF : DD_k \cup SRS_k \cup \widehat{CONF}$ *is satisfiable* $\forall k \in \{1, \ldots, n\}$. Finally we have to show the algorithm terminates. Sources for infinite processing loops are cycles in message passing and subsequent generation of the same conflict. Infinite processing loops are not possible because all agents can only receive requests from the facilitator that are replied either by a $reply^{no}_k$ or a $conflict_k$ message, and the number of these messages is restricted due to the initial assumption of a limited solution space. Subsequent generation of the same conflict is avoided, because inconsistent configurations are

distributed as conflicts[4] to all agents via $add\text{-}conflict$ messages and locally stored by them. As all valid configurations are consistent with the set of already negated conflicts, no conflicting configuration is produced twice by a configuration agent. Further with this algorithm all valid configurations can be found. If an already found solution is negated and added to the system requirements, a new search can be initiated and the algorithm will find another solution that is different from the previous one. As there exists only a finite number of configurations, subsequently all solutions will be found.

5 Aspects of Integration

In order to be integrated in the presented architectural setting, a configurator must satisfy the following requirements resulting from the protocol:

- Support of $request_k^{no}$ requests: The configurator must accept a partial configuration as starting point for configuration problem solving and generate a complete configuration solution.
- Support for $add\text{-}conflict$ messages: The configurator must at least be able to compute alternative solutions to fulfill this requirement. An agent wrapper would then store the received conflicts and request alternative solutions from the native configurator interface, until all stored negated conflicts are satisfied.

The latter requirement can not be met by pure rule-based configurators, which always calculate only exactly one solution for given requirements. Furthermore, different expressiveness of proprietary knowledge representations may pose a problem. Employing bridging rules, that map between different representation concepts are always imperfect in the sense that they are heuristic.
A different issue is the process of distributedly creating and maintaining configuration knowledge. There must be some guidance provided and the consistency of the knowledge bases has to be assured. Having different representation mechanisms, a shared ontology must be adopted, that provides a common view on the product domain.
Currently, separate sets of initial requirements SRS_k are assumed. However, the choice among similar products of different companies is another point to be addressed and some form of reasoning on the selection of a supplier has to be introduced.

6 Related Work

There is a long history in developing configuration tools in knowledge-based systems. Progressing from rule-based systems higher level representation formalisms were developed, such as various forms of constraint satisfaction [4], or description logics [8]. However there is no support for integrating these systems in order to allow cooperative configuration.
When using a constraint-based approach for configuration tasks, several techniques

[4] Note that Skolem constants in calculated configurations are converted to all-quantified variables, if the conflict is negated.

for distributed problem solving have been proposed. For problem representation a distributed CSP is proposed in [13] and several algorithms for problem solving such as an *asynchronous backtracking*, an *asynchronous weak-commitment* search or a *distributed breakout* algorithm are presented. However, configuration tasks are more dynamic in nature and therefor a CSP representation, where all problem variables must be known from the beginning, is not appropriate in many application domains. Dynamic constraint satisfaction is more suitable for representing and solving such synthesis tasks [9] [12], because the set of problem variables may vary according to some *activity contraints*. In [3] a distributed dynamic CSP is defined and a modification of the asynchronous backtracking algorithm from [13] is applied for problem solving. When configuring large technical systems, the limitation of a dynamic CSP representation (the amount of maximally active variables must be known from the beginning) becomes evident and a generic CSP representation [4] [7] has been proposed. There, new instances of problem variables can be created from meta-variables during problem solving. However, no representation that allows the distribution of knowledge over several agents has been presented so far.

The proposed architecture for distributed configuration relates to previous research projects such as TSIMMIS [5] or Infomaster [6]. They provide an integrated access to multiple distributed heterogenous information sources on the Internet. Our approach for distributed configuration goes a step further, because not only information sources but problem-solving agents with local knowledge are integrated, thus giving the illusion of a centralized, homogenous configuration system.

In the area of distributed configuration-design problem solving [1] proposed an agent architecture. The aim of this work is to find a concurrent problem solving process in order to improve efficiency, whereas our concern is to provide effective support of distributed configuration problem solving, where knowledge is already distributed between different agents.

7 Conclusions

Due to internet technologies, business processes cross enterprise boundaries, which boosts the demand for distributed problem solving methods. In the domain of product configuration the integration of Web-based configuration agents is necessary in order to match the needs that arise from temporary cooperation between highly specialized business entities. In this paper we defined a general consistency-based approach towards the joint provision of configuration solutions by multiple configurators. Based on a formal definition of the distributed configuration approach, it was shown under which conditions distributed configuration problem solving produces equivalent results to the central case. Partial configurations which are in conflict to the system requirements and domain descriptions facilitate the search process. The concept of conflicts was introduced and its relation to valid configurations shown. Further a complete and sound algorithm for cooperation was presented which allows the integration of domain dependent heuristics.

Acknowledgments. This work was partly funded by the EC under contract no. IST-1999-10688 and the Austrian Federal Ministry of Education, Science and Culture.

References

1. T.P. Darr and W.P. Birmingham. An Attribute-Space Representation and Algorithm for Concurrent Engineering. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 10(1):21–35, 1996.
2. A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based Diagnosis of Configuration Knowledge Bases. In *Proceedings of the 14th ECAI*, pages 146–150, Berlin, Germany, 2000.
3. A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Distributed Configuration as Distributed Dynamic Constraint Satisfaction. In *Proceedings of the 14th IEA/AIE*, pages 434–444, Budapest, Hungary, 2001.
4. G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. In B. Faltings and E. Freuder, editors, *IEEE Intelligent Systems, Special Issue on Configuration*, volume 13,4, pages 59–68. 1998.
5. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.
6. M. Genesereth, A. Keller and O. Duschka. Infomaster: An Information Integration System. In *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, pages 539–542, Tucson, Az, USA, 1997.
7. D. Mailharro. A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Special Issue: Configuration design*, 12(4):383–397, 1998.
8. D.L. McGuiness and J.R. Wright. Conceptual Modeling for Configuration: A Description Logic-based Approach. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Special Issue: Configuration design*, 12(4):333–344, 1998.
9. S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proceedings of AAAI 1990*, pages 25–32, Boston, MA, 1990.
10. S. Mittal and F. Frayman. Towards a Generic Model of Configuration Tasks. In *Proc. of the 11th IJCAI*, pages 1395–1401, Detroit, MI, 1989.
11. D. Sabin and R. Weigel. Product Configuration Frameworks - A Survey. In E. Freuder B. Faltings, editor, *IEEE Intelligent Systems, Special Issue on Configuration*, volume 13(4), pages 50–58. 1998.
12. T. Soininen, E. Gelle and I. Niemelä. A Fixpoint Definition of Dynamic Constraint Satisfaction. In *5th International Conference on Principles and Practice of Constraint Programming - CP'99*, pages 419–433, Alexandria, USA, 1999.
13. M. Yokoo. *Distributed Constraint Satisfaction - Foundations of Cooperation in Multi-agent Systems*. Springer, Berlin, Germany, 2001.

Belief Update in the pGOLOG Framework

Henrik Grosskreutz and Gerhard Lakemeyer

Department of Computer Science V
Aachen University of Technology
D-52056 Aachen, Germany
{grosskreutz,gerhard}@cs.rwth-aachen.de

Abstract. High-level controllers that operate robots in dynamic, uncertain domains are concerned with at least two reasoning tasks dealing with the effects of noisy sensors and effectors: They have a) to project the effects of a candidate plan and b) to update their beliefs during on-line execution of a plan. In this paper, we show how the pGOLOG framework, which in its original form only accounted for the projection of high-level plans, can be extended to reason about the way the robot's beliefs evolve during the on-line execution of a plan. pGOLOG, an extension of the high-level programming language GOLOG, allows the specification of probabilistic beliefs about the state of the world and the representation of sensors and effectors which have uncertain, probabilistic outcomes. As an application of belief update, we introduce *belief-based programs*, GOLOG-style programs whose tests appeal to the agent's beliefs at execution time.

1 Introduction

High-level robot controllers that operate in dynamic, uncertain domains and have to cope with sensors and effectors that have uncertain, probabilistic outcomes are concerned with at least two distinct reasoning tasks. First, given a candidate plan, *probabilistic projection* allows the prediction of the effects of the plan. This task, which is a prerequisite to deliberate over different possible plans, lies at the heart of probabilistic planning [KHW95, DHW94] and the theory of POMDPs [KLC98]. Second, given a characterization of the robot's beliefs about the state of the world, a robot should be able to update its beliefs during the execution of a plan interacting with the robot's noisy sensors and effectors. This second task, to which we will refer to as *belief update*, following [BHL99], is necessary to allow probabilistic reasoning (in particular probabilistic projection) in non-initial situation.

In this paper, we show how the probabilistic high-level programming framework pGOLOG [GL00], which in its original form only accounted for the probabilistic projection of high-level plans interacting with noisy sensors and effectors, can be extended to reason about the way the robot's beliefs evolve during the on-line execution of a plan.[1] To do so, we first explicitly model a layered robot control architecture where the robot's high-level controller does not directly affect the world by operating the robot's physical sensors and effectors, but instead is connected to a basic-task execution level which provides specialized *low-level processes* like navigation, object recognition or grasping

[1] Here, we use the term on-line-execution in the sense of [dGL99].

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 213–228, 2001.

objects. Having such a model has the advantage that there is a clear separation of the actions of the high-level controller from those of the low-level processes. In particular, while the action of activating a low-level process and its execution time are under the control of the high-level controller, neither are the effects of the activated process nor its completion time.

To model the effects of the low-level processes, we make use of probabilistic programs, where the different probabilistic branches of the programs correspond to different possible outcomes of the low-level processes. Modelling low-level processes as programs allows a very fine-grained characterization of the effects of the low-level processes at a level of detail involving many atomic actions, taking into account the temporal extent of the processes. Based on such a model of the possible effects of the low-level processes, we specify how the robot's beliefs about the state of the world evolve during the on-line execution of a plan, in particular how the beliefs change when the robot activates a low-level process that operates the robot's physical effectors or when a low-level process provides noisy information about the state of the world. Finally, we show how based on the robot's evolving belief state it becomes possible to execute so-called *belief-based programs*, GOLOG-style programs [GLL00] whose tests appeal to the agent's beliefs *at execution time*.

To get a better feel for what we are aiming at, let us consider the following *ship/reject*-example, adapted from [DHW94]: We are given a manufacturing robot with the goal of having a widget painted (*PA*) and processed (*PR*). Processing widgets is accomplished by rejecting parts that are flawed (FL) or shipping parts that are not flawed. Initially, the probability of being flawed is 0.3. *ship* and *reject* always make PR TRUE, however *ship* causes an execution error (ER) if FL holds, and *reject* causes ER to be TRUE if FL does not hold. The robot can activate a low-level process *paint*, which first under-coats the widget (UC) for 10 seconds, then takes 20 seconds to paint it. However, *paint* has a 5% probability to fail. There is also a low-level process *inspect* which can be used to determine whether or not the widget is flawed. However, *inspect* has a 10% probability to overlook a flaw and report *OK* instead of $\overline{OK}$ even though the widget is flawed; if the widget is not flawed, it always reports *OK*.

In this scenario, an example projection task is: how probable is it that the plan "first *inspect* the widget; thereafter, if *OK* holds then *ship* else *reject* it" will falsely ship a flawed widget. On the other hand, belief update is concerned with questions like: what is the probability that the widget is flawed if during on-line-execution the robot actually perceived *OK* (the respective probabilities are 0.3*0.1=3% and 3/73=4.1%). The difference between the two tasks is that in the former case, the agent reasons about how the world might evolve, while in the latter case its beliefs change as a result of actual actions. We remark that besides updating its beliefs concerning the state of the world in terms of fluents like *PA* or *PR*, the robot also has to update its beliefs concerning the state of execution of the low-level processes; for example, 15 seconds after activation of the *paint* process the robot should not only be aware of the fact that the widget is under-coated by now, but also that the process is no longer in its initial state but only 15 seconds away from completion. Finally, a belief-based plan is a specification appealing to the robot's beliefs at execution time, like for example "as long as your (i.e. the robot's) confidence in whether the widget is flawed or not is below a threshold of 99%, (re-)inspect the widget.

Thereafter, ship the widget if your belief in the widget being not flawed exceeds 99%, else reject it." Note that in this plan the activation of low-level processes is conditioned on the robot's beliefs at execution time.

The rest of this paper is organized as follows: after a brief review of the situation calculus and pGOLOG, we describe an overall robot control architecture for acting under uncertainty. Thereafter, we specify how the robot's probabilistic belief state evolves during the course of action. Finally, we introduce belief-based programs and show how they can be used to solve the example problem. The paper ends with a discussion of related work and concluding remarks.

2 The Situation Calculus

We will only go over the situation calculus [McC63,LPR98] briefly here: all terms in the language are of sort ordinary objects, actions, situations, or reals.[2] There is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation of s resulting from performing action a in s; relations and functions whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate resp. function symbols taking a situation term as their last argument; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s. Within this language, we can formulate theories which describe how the world changes as the result of the available actions. One possibility is a *basic action theory* of the following form [LPR98]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms (SSA), one for each fluent F, stating under what conditions $F(\boldsymbol{x}, do(a, s))$ holds as a function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem.
- Domain closure and unique name axioms for the primitive actions.
- A collection of foundational, domain independent axioms. One of them defines how a situation s' can be reached from a situation s by a sequence of actions:[3]
  $$s \sqsubset do(a, s') \equiv s \sqsubseteq s'$$
  where $s \sqsubseteq s'$ stands for $(s \sqsubset s') \vee (s = s')$.

Adding a Timeline. In its basic form, the situation calculus has no notion of time. In order to model that processes have temporal extent, we introduce a special unary functional fluent *start* of sort real. The understanding is that $start(s)$ denotes the time when situation s begins (we assume that $start(S_0)$ is 0). The fluent *start* changes its value only as a result of the special primitive action $tUpdate(t)$, with the intuition that,

[2] While the reals are not normally part of the situation calculus, we need them to represent probabilities. For simplicity, the reals are not axiomatized and we assume their standard interpretations together with the usual operations and ordering relations.

[3] We use the convention that all free variables are implicitly universally quantified.

normally, every action is instantaneous, that is, the starting time of the situation after doing a in s is the same as the starting time of s. The only exception is *tUpdate*(t). Whenever this action occurs, the starting time of the resulting situation is advanced up to t. The following axiom makes this precise.

$$Poss(a, s) \supset [\mathit{start}(do(a, s)) = t \equiv$$
$$a = \mathit{tUpdate}(t) \vee t = \mathit{start}(s) \wedge \neg\exists t'.a = \mathit{tUpdate}(t')$$

We will see in Section 4 how *tUpdate* actions are used to synchronize *start* with the actual time during on-line execution of robot plans. We remark that in [GL01] a version of the temporal situation calculus is considered where it is possible to wait for conditions like a robot arriving at a certain location, which is modeled using continuous functions of time, an issue we ignore here for simplicity.

3 pGOLOG

As argued in [GL00], robot "actions" such as *paint* or *inspect* are often best thought of as low-level processes with uncertain, probabilistic outcome which need to be described at a level of detail involving many atomic actions, rather than as primitive, atomic actions. To describe such processes, we proposed to model the processes as programs using the probabilistic language pGOLOG. The idea is to *model the noisy low-level processes as probabilistic programs*, where the different probabilistic branches of the programs correspond to different possible outcomes of the processes. Given a faithful characterization of the low-level processes in terms of pGOLOG programs, we can then reason about the effect of the activation of the processes through simulation of their corresponding pGOLOG models.

Besides constructs such as sequences, iterations and recursive procedures, pGOLOG provides a probabilistic branching instruction: *prob*(p, σ_1, σ_2). Its intended meaning is to execute program σ_1 with probability p, and σ_2 with probability $1-p$. In addition to the constructs already present in [GL00], we introduce the parallel construct *withPol*(σ_1, σ_2) adapted from [GL01]. The intuition is to execute σ_1 and σ_2 concurrently until σ_2 ends. The program σ_1 has a higher priority than σ_2, meaning that whenever both σ_1 and σ_2 are about to execute an action at the same time, σ_1 takes precedence. We remark that pGOLOG only provides deterministic instructions.

α	primitive action
$\phi?$	wait/test action[4]
$[\sigma_1, \sigma_2]$	sequence
$if(\phi, \sigma_1, \sigma_2)$	conditional
$while(\phi, \sigma)$	loop
$prob(p, \sigma_1, \sigma_2)$	probabilistic execution
withPol(σ_1, σ_2)	prioritized execution until σ_2 ends
proc$(\beta(\boldsymbol{x})\ \sigma)$	procedure definition

[4] Here, a condition ϕ stands for a situation calculus formula where *now* may be used to refer to the current situation; when no confusions arise, we will simply leave out the *now* argument from the fluents altogether. Similarly, the term $\phi[s]$ denotes the formula obtained by substituting the situation variable s for all occurrences of *now* in fluents appearing in ϕ.

To illustrate the use of pGOLOG, we will now model the possible effects of *paint* by the pGOLOG program *paintProc*. Intuitively, if the widget is already processed, trying to *paint* it results in an error. Otherwise, 10 seconds after activation of *paint* the widget will become under-coated, and finally after 30 seconds *paint* will result in the widget being painted with probability 95 % (there is also a 5 % chance that *paint* will remain effectless). To model the effects of *paint*, we make use of the fluents *PA*, *FL*, *PR* and *ER* with the obvious meaning to represent the properties of our example domain, and assume successor state axioms that ensure that the truth value of *PA* is only affected by the primitive actions *setPA* and *clipPA*, whose effect is to make it TRUE resp. FALSE; similarly for the other fluents.

$$\begin{aligned} &\mathit{proc}(\mathit{paintProc}, [\mathit{waitTime}(10), \mathit{if}(\mathit{PR}, \mathit{setER}, \mathit{setUC}),\\ &\qquad\qquad \mathit{waitTime}(20), \mathit{if}(\mathit{PR}, \mathit{setER}, \mathit{prob}(0.95, \mathit{setPA}))]). \end{aligned}$$

Here, $\mathit{waitTime}(n)$ is a procedure whose purpose is to wait for n seconds. It essentially corresponds to a test $\mathit{start} \geq \tau + n?$, where τ refers to the time where *waitTime* was invoked. We will see in the Section 5 how this and similar pGOLOG models of the robot's low-level processes are used to update the robot's beliefs during on-line execution.

Formal Semantics. The semantics of pGOLOG is defined using a so-called transition semantics similar to ConGolog [GLL00]. It is based on defining single steps of computation and, as we use a probabilistic framework, their relative probability. There is a function $transPr(\sigma, s, \delta, s')$ which, roughly, yields the transition probability associated with a given program σ and situation s as well as a new situation s' that results from executing σ's first primitive action in s, and a new program δ that represents what remains of σ after having performed that action.[5] Furthermore, there is another predicate $\mathit{Final}(\sigma, s)$ which specifies which configurations (σ, s) are final, meaning that the computation can be considered completed. This is the case, roughly, when the remaining program is *nil*, but not if there is still a primitive action or test action to be executed.

For space reasons, we only list a few of the axioms for $transPr$ and *Final*. Let us first look at *withPol* and *prob* informally: the execution of σ_2 with policy σ_1 means that one action of one of the programs is performed, whereby actions which can be executed earlier are always preferred. If both σ_1 and σ_2 are about to execute an action at the same time, the policy σ_1 takes precedence. The whole *withPol* construct is completed as soon as σ_2 is completed. The execution of $\mathit{prob}(p, \sigma_1, \sigma_2)$ results in the execution of a dummy, i.e. effectless action *tossHead* or *tossTail* with probability p resp. $1 - p$ with remaining program σ_1, resp. σ_2. Let nil be the empty program and α a primitive action.

$$\begin{aligned} &transPr(nil, s, \delta, s') = 0\\ &transPr(\alpha, s, \delta, s') =\\ &\quad \textbf{if } Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s) \textbf{ then } 1 \textbf{ else } 0\\ &transPr([\sigma_1, \sigma_2], s, \delta, s') =\\ &\quad \textbf{if } \delta = [\delta', \sigma_2] \textbf{ then } transPr(\sigma_1, s, \delta', s') \end{aligned}$$

[5] Note that the use of a transition semantics necessitates the reification of programs as first order terms in the logical language, an issue we gloss over completely here (see [GLL00] for details). For space reasons, we also completely gloss over the definition of *proc*, which requires a second order definition of $transPr$.

$$\begin{array}{l}
\quad \textbf{else if } \mathit{Final}(\sigma_1, s) \textbf{ then } \mathit{transPr}(\sigma_2, s, \delta, s') \textbf{ else } 0 \\
\mathit{transPr}(\mathit{prob}(p, \sigma_1, \sigma_2), s, \delta, s') = \\
\quad \textbf{if } \delta = \sigma_1 \wedge s' = \mathit{do}(\mathit{tossHead}, s) \textbf{ then } p \textbf{ else} \\
\quad\quad \textbf{if } \delta = \sigma_2 \wedge s' = \mathit{do}(\mathit{tossTail}(\mathit{start}, s) \textbf{ then } 1 - p \textbf{ else } 0 \\
\mathit{transPr}(\mathit{withPol}(\sigma_1, \sigma_2), s, \delta, s') = \\
\quad \textbf{if } \exists \delta_1 . \delta = \mathit{withPol}(\delta_1, \sigma_2) \wedge \mathit{transPr}(\sigma_1, s, \delta_1, s') > 0 \wedge \\
\quad\quad \neg \mathit{Final}(\sigma_2) \wedge \forall \delta_2, s_2 . \mathit{transPr}(\sigma_2, s, \delta_2, s_2) > 0 \supset \\
\quad\quad\quad \mathit{start}(s') \leq \mathit{start}(s_2) \textbf{ then } \mathit{transPr}(\sigma_1, s, \delta_1, s') \\
\quad \textbf{else if } \exists \delta_2 . \delta = \mathit{withPol}(\sigma_1, \delta_2) \wedge \\
\quad\quad \mathit{transPr}(\sigma_2, s, \delta_2, s') > 0 \wedge \forall \delta_1, s_1 . \\
\quad\quad\quad \mathit{transPr}(\sigma_1, s, \delta_1, s_1) > 0 \supset \mathit{start}(s') < \mathit{start}(s_1) \\
\quad\quad \textbf{then } \mathit{transPr}(\sigma_2, s, \delta_2, s') \textbf{ else } 0
\end{array}$$

$$\begin{array}{ll}
\mathit{Final}(\alpha, s) \equiv \textsc{False} & \mathit{Final}(\mathit{prob}(p, \sigma_1, \sigma_2), s) \equiv \textsc{False} \\
\mathit{Final}(\mathit{nil}, s) \equiv \textsc{True} & \mathit{Final}(\mathit{withPol}(\sigma_1, \sigma_2), s) \equiv \mathit{Final}(\sigma_2, s)
\end{array}$$

So far, we have only defined which successor configurations can be reached through a single transition. The predicate $doPr(\sigma, s, s')$ defines the probability of an execution trace s' of program σ starting in s, that is the probability to end up in a final configuration with situation component s' after a *sequence* of transitions. In the following axiom, $transPr^*(\delta, s, \delta', s')$ refers to the transitive closure of $transPr$.

$$\begin{array}{l}
doPr(\delta, s, s') = \\
\quad \textbf{if } \exists \delta', p . p > 0 \wedge transPr^*(\delta, s, \delta', s') = p \wedge \mathit{Final}(\delta', s') \textbf{ then } p \textbf{ else } 0
\end{array}$$

Intuitively, if $\langle \delta', s' \rangle$ can be reached from $\langle \delta, s \rangle$, then $transPr^*(\delta, s, \delta', s')$ is the product of the probabilities of each transition along the path from $\langle \delta, s \rangle$ to $\langle \delta', s' \rangle$. For space reasons, we omit the definition of the transitive closure of $transPr$ (which requires second order logic) and refer the interested reader to [GL00].

4 A Control Architecture for Acting under Uncertainty

In modern robot control architectures like RHINO [BCF+00], the robot's high-level controller does not directly affect the world by operating the robot's physical sensors and effectors, but instead is connected to a basic-task execution level which provides specialized low-level processes like navigation, object recognition or grasping objects. We will now describe how this type of architecture can be reconstructed in a logic-based framework; the architecture presented here is essentially an extention of [GL01], adapted to stochastic scenarios. In particular, we allow for the robot's uncertainty about the state of the world, account for the fact that low-level processes have uncertain outcomes, and show how to deal with processes like *inspect* which provide information about the state of the world. The resulting overall architecture is illustrated in Figure 1.

In order to reason about the effects of a high-level plan, we need a model of every part of the robot control architecture illustrated in Figure 1. (A robot controller that lacks a model of the effects of its actions is intrinsically incapable to reason about the effects of its actions). Let us start with a representation of the state of the world.

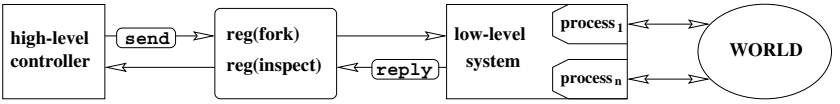


Fig. 1. Robot Control Architecture for Acting under Uncertainty

4.1 The State of the World

While the original situation calculus allows us to talk only about the actual state of the world, in scenarios like the ship/reject example we have to represent uncertain beliefs about the state of the world. To do so, we follow [BHL99] and characterize the probabilistic epistemic state of a robot by a *set of situations considered possible*, and the *likelihood* assigned to the different possibilities. More specifically, there is a binary functional fluent $p(s', s)$ which can be read as "in situation s, the agent thinks that s' is possible with weight $p(s', s)$.[6]" All weights must be non-negative and situations considered impossible will be given weight 0 (we do not require that the weights sum to 1). Furthermore, all situations considered possible in S_0 must be initial.

$$\forall s'.p(s', S_0) > 0 \supset \forall s'', a''.s' \not= do(a'', s'')$$

For example, in the introductory ship/reject domain the world is in one of two states, $s1$ and $s2$, which occur with probability 0.3 and 0.7, respectively. All other situations have likelihood 0. The following axiom makes this precise together with what holds and does not hold in each of the two states.

$$\begin{array}{l}\forall s.p(s, S_0) > 0 \supset \neg PA(s) \wedge \neg PR(s) \wedge \neg ER(s) \wedge \\ \exists s_1, s_2.p(s_1, S_0) = 0.3 \wedge p(s_2, S_0) = 0.7 \wedge \\ \quad FL(s_1) \wedge \neg FL(s_2) \wedge \forall s.s \not= s_1 \wedge s \not= s_2 \supset p(s, S_0) = 0\end{array}$$

Belief. Based on p, [BHL99] define $Bel(\phi, s)$, the agent's degree of belief that ϕ holds in situation s, to be an abbreviation for the following term expressible in second-order logic.[7]

$$\Sigma_{\{s':\phi[s']\}} p(s', s) / \Sigma_{s'} p(s', s)$$

Intuitively, $Bel(\phi, s)$ is the normalized sum of the weights of all situations s' considered possible in s that fulfill ϕ. In our example, $Bel(FL, S_0)$ is 0.3.

4.2 Communication between Low-Level Processes and High-Level Controller

We assume that the entire communication between the high-level controller and the low-level processes is achieved through a set of registers, and model them by the special functional fluent $reg(id, s)$. The high-level interpreter can affect the value of *reg* by means of the special action $send(id, val)$ which assigns $reg(id, s)$ the value val. The intuition is that in order to activate a low-level process, the high-level controller executes a *send*

[6] Having more than one initial situation means that Reiter's induction axiom for situations [LPR98] no longer holds, just as in [BHL99].

[7] As before, ϕ is a situation calculus formula where *now* is used to refer to the current situation.

action. For example, the execution of $send(fork, paint)$ would tell the execution system to start the paint process.[8]

On the other hand, the low-level processes can provide the high-level controller with sensor information by means of the exogenous[9] action $reply(id, val)$. The following successor state axiom specifies how *reg* changes its value.

$$Poss(a, s) \supset [reg(id, do(a, s)) = val \equiv a = send(id, val) \vee a = reply(id, val) \vee reg(id, s) = val \wedge \neg(\exists r, v. a = send(r, v) \vee a = reply(r, v))]$$

We assume that initially, the value of the fluent *reg* is *nil* for all *id*, and that the robot know about this.

$$\forall id. reg(id, S_0) = nil \wedge \forall s, id. p(s, S_0) > 0 \supset reg(id, s) = reg(id, S_0)$$

4.3 The Low-Level Execution System

Next, let us model the low-level execution system, starting with the individual low-level processes. As mentioned in Section 3, we model all low-level processes by probabilistic pGOLOG programs. While we have already modeled *paint* by the procedure *paintProc*, we model *ship* and *reject* by the following two pGOLOG programs. We assume that both processes take 10 seconds to complete execution, whereupon they confirm completion by means of a $reply(processed, t)$ action.

$$proc(shipProc, [waitTime(10), if(PR \vee FL, setER), setPR, reply(processed, t)])$$
$$proc(rejectProc, [waitTime(10), if(PR \vee \neg FL, setER), setPR, reply(processed, t)])$$

Sensor Processes. Next, we turn to the process *inspect*. At this point, we have to explain what we mean by sensing. To us, sensing means: activate a sensor. This "activation" has as an effect a sensor reading. In the example, sensing happens through the activation of the *inspect* process, whose effect is to provide a $reply(inspect, OK)$ or $reply(inspect, \overline{OK})$ answer. We assume that the high-level controller is aware of all exogenous *reply* actions, as opposed to "actions" like *setPA* which are solely used to model the effects of the low-level processes. Sensing is thus realized by means of special low-level processes, which we call *sensor processes* and which communicate (pre-processed) sensor readings by means of exogenous *reply* actions. Note that this view of sensing significantly differs from the well-known sensing actions of [Lev96].

Although during real execution the actual low-level process *inspect* provides the answer, we need a model of the behavior of the sensor to update the robot's beliefs after *OK* or $\overline{OK}$ answers. The following pGOLOG program describes the possible effects of *inspect*. We use *replyOK* as an abbreviation for $reply(inspect, OK)$, similarly for $reply\overline{OK}$.

$$proc(inspectProc, if(FL, [waitTime(10), prob(0.9, reply\overline{OK}, replyOK)], [waitTime(10), replyOK]))$$

[8] The term *fork* refers to the procedure `fork` used in UNIX- like operating systems to create new concurrent processes.

[9] Here, an exogenous action is an action not under the control of the high-level controller.

Directly Observables. *reply* actions like the above provide the high-level controller with information because they assign $reg(inspect, s)$ a value which is correlated with the value of *FL* (i.e. *OK* or $\overline{OK}$) and because unlike in the case of *FL* there is no uncertainty about the value of *reg*. We distinguish *reg* from other fluents and call it *directly observable*, following [GL00]. Directly observable fluents are such that the agent always has perfect information about them - like the display of one's watch or a fuel gauge in the car. Formally, we call a relational fluent P directly observable wrt a pGOLOG theory iff. the following formula holds:

$$\forall s, s', \boldsymbol{x}.\ S_0 \prec s \wedge p(s', s) > 0 \supset P(s', \boldsymbol{x}) \equiv P(s, \boldsymbol{x})$$

Directly observable functional fluents are defined similarly. We remark that the initial and successor state axioms for *reg* presented in this section together with the successor state axiom for p in the following section guarantee that in our example *reg* is in fact directly observable.

The Overall Low-Level Execution System. Finally, we need a formal model of the execution system as a whole, i.e. of the robots operating system, which ensures that *send* actions result in the activation of the corresponding low-level process. The following program *kernelProc* describes the "kernel process" of the robot's operating system.

$$\begin{array}{l} proc(kernelProc, [reg(fork)\ /\!=\!nil?, \\ \quad if(reg(fork) = inspect, [reply(fork, nil), withPol(inspectProc, kernelProc)], \\ \quad\quad if(reg(fork) = paint, [reply(fork, nil), withPol(paintProc, kernelProc)], \\ \quad\quad\quad ...,\ \textbf{else}\ [reply(fork, nil), kernelProc])...)]) \end{array}$$

As long as $reg(fork)$ is nil, nothing happens. If $reg(fork)$ is assigned the name of a low-level process, then $reg(fork)$ is reset to nil, and the low-level process is run concurrently to the operating system's kernel process. We stress that pGOLOG programs such as the above are not intended for actual execution. Their purpose is to provide a *model* of the behavior of the low-level process. In fact, pGOLOG programs like *inspectProc* or *paintProc* cannot be executed by the high-level controller because it has only uncertain information about the value of non-observable fluents like *FL*, resp. because it cannot directly execute actions like *setPA*.

4.4 The High-Level Controller

In order to ensure that the high-level controller will always have the necessary knowledge to evaluate tests within high-level robot plans, we consider only a subset of the pGOLOG programs as legal high-level plans. This subset of pGOLOG, to which we refer to as GOLOG_{rp}, consists of all programs whose tests are restricted to *directly observable fluents*, and which only execute actions that *only affect directly observables*. We gloss over the technical details. As an example, the following GOLOG_{rp} plan activates both *inspect* and *paint*, waits for their completetion and finally processes the widget according to the result of *inspect*. We use *OK* as an abbreviation for $reg(inspect, OK)$, *forkInspect* as an abbreviation for $send(fork, inspect)$, and similarly for *forkPaint*, *forkShip* and *forkReject*.

$$\begin{array}{l} proc(Prg_{ex}, [forkPaint, waitTime(30), forkInspect, reg(inspect)\ /\!=\!nil?, \\ \qquad\qquad if(OK, forkShip, forkReject), reg(processed)\ /\!=\!nil?]) \end{array}$$

We remark that during on-line execution of a GOLOG$_{rp}$ plan, whenever the high-level plan executes a *send* action, the interpreter checks whether this signals an activation of a low-level process and, as a side-effect, activate the actual low-level process if necessary.

The Passage of Time during On-line Execution. Finally, a word on the passage of time during on-line execution of a high-level plan. In order to synchronise the internal clock, i.e. the value of the fluent *start* with the actual time during on-line execution, the high-level controller periodically generated exogenous *tUpdate*(t) events, where t refers to the actual time. As described in Section 2, the effect of a *tUpdate* is then to assign *start* the actual time. We assume that the difference $\Delta = t_{i+1} - t_i$ between two subsequent updates *tUpdate*(t_i) and *tUpdate*(t_{i+1}) is smaller than the minimal delay between the execution time of any two actions of the pGOLOG models which have different execution time. Furthermore, we assume that if a *reply* is modeled to happen at time t, then during on-line execution the high-level controller will generate a *tUpdate* action causing *start* to advance to t before the actual *reply* action happens.

5 Belief Update

We now have a model of the robot's control architecture, of its beliefs about the state of the world, and of the execution system of the robot including models of the low-level processes. Based on this model, we will now specify how to update the robot's belief state as a result of the activation of noisy low-level processes and of the receipt of *reply* messages. We refer to this task as to *(probabilistic) belief update*, following [BHL99].

Although not quite obvious, the specification of a successor state axiom for the fluent p is not sufficent to represent the updated belief state. To see why, let us consider the situation S_{uc} where the robot has activated the *paint* process in the initial situation through *send*(*fork*, *paint*), after which it has waited for 15 seconds. Intuitively, the epistemic state should reflect the fact that the activation of the low-level process *paint* has affected the truth value of *UC*. But this is not sufficient. Additionally, the robot should be aware of the fact that unlike in S_0, in S_{uc} the low-level process *is active*, has already executed *setUC*, and is about to probably execute *setPA*. Thus, the *paint* process is no longer correctly characterized by *paintProc*, but instead by the remaining fragment of *paintProc* after 15 seconds have passed.

The example suggests that the appropriate pGOLOG model of the low-level processes is not the same for all situations, but depends on the history of actions. Thus, we associate with every possible situation a specific pGOLOG model. Formally, we introduce a special functional fluent $ll(s', s)$ that can be read as "in situation s, the robot thinks that if the world is in situation s' then the low-level processes can be characterized by the pGOLOG program $ll(s', s)$." The following axiom states that in the initial situation the low-level processes are as described by *kernelProc* (defined above).

$$\forall s.p(s, S_0) \supset ll(s, S_0) = \mathit{kernelProc}.$$

In order to specify successor state axioms for $p(s^*, do(a, s))$ and $ll(s^*, do(a, s))$, stating how the world and the low-level processes evolve from a situation s to its successor situation $do(a, s)$, we have to distinguish two cases: (i) a is a *reply* action performed by a

sensor process; and (ii) a is an actions executed by the high-level controller or a *tUpdate* action. The reason that we have to distinguish *reply* actions from other, "ordinary" actions is that *reply* actions provide sensing information, as captured by the pGOLOG model of the sensing processes (like, for example, *inspectProc*). Note that, as stated above, we assume that the high-level controller is not aware of any "action" performed by the low-level processes except for the *reply* actions.

5.1 Ordinary Actions

Let us first consider the second case. Our solution is that the low-level processes execute up to the point where one of the following conditions occur:

1. they are blocked, i.e. waiting for a $\phi?$ condition to become true;
2. or they are about to execute an *reply* action.

While the first condition is fairly obvious, the reason that we mind *reply* actions is that the high-level controller is aware of all *reply* actions, and a is no *reply* action. We will now formalize the idea to execute a program σ in s until a configuration $\langle \delta, s' \rangle$ is reached where one of the above conditions is true. For this, we use the special function $transPr^{\triangleleft}(\sigma, s, \delta, s')$ which specifies the probability to end up in $\langle \delta, s' \rangle$ starting in $\langle \sigma, s \rangle$. In the following formulas, $\mathfrak{R}(a)$ is a shorthand for $\exists r, v.a = \mathit{reply}(r, v)$.

$$
\begin{array}{l}
transPr^{\triangleleft}(\sigma, s, \delta, s') = \\
\quad \textbf{if } transPr^{*}(\sigma, s, \delta, s') > 0 \wedge \forall a^*, s^*.s \sqsubset do(a^*, s^*) \sqsubseteq s' \supset \neg\mathfrak{R}(a^*) \wedge \\
\qquad \forall \delta^*, s^*.transPr(\delta, s', \delta^*, s^*) > 0 \supset \exists a^*.s^* = do(a^*, s') \wedge \mathfrak{R}(a^*) \\
\quad \textbf{then } transPr^{*}(\sigma, s, \delta, s') \textbf{ else } 0
\end{array}
$$

While the first line of the **if** condition verifies that $\langle \delta, s' \rangle$ can be reached from $\langle \sigma, s \rangle$ without executing any *reply* action, the second line verifies that all successor configurations of $\langle \delta, s' \rangle$ can only be reached by a violation of the second of the above conditions, meaning that the simulation has been pursued as far as possible.

Using $transPr^{\triangleleft}$, we can define which configurations $\langle s^*, ll^* \rangle$ have been reached by the low-level processes in $do(a, s)$ together with their weight (assuming that a is no *reply* action). Intuitively, these are all configuration that result from the execution via $transPr^{\triangleleft}$ of a configuration $\langle ll(s', s), s' \rangle$ considered possible in s. Their weight is the product of the weight of s' in s and the transition probability as specified by $transPr^{\triangleleft}$. The predicate $\mathit{advConfig}(s^*, ll^*, do(a, s))$ makes this precise.

$$
\begin{array}{l}
\mathit{advConfig}(s^*, ll^*, do(a, s)) = p \equiv \\
\quad \exists s', p', p^*.p(s', s) = p' \wedge transPr^{\triangleleft}(ll(s', s), s', ll^*, s^*) = p^* \wedge \\
\quad p' > 0 \wedge p^* > 0 \wedge p = p' \cdot p^* \vee \\
\quad p = 0 \wedge \neg\exists s'.p(s', s) > 0 \wedge transPr^{\triangleleft}(ll(s', s), s', ll^*, s^*) > 0
\end{array}
$$

5.2 *reply* Actions

Now that we have formalized how the low-level processes evolve if a is an ordinary action, let us turn to the other case where a is a *reply* action. Intuitively, the observation of a *reply* should sharpen the belief state of the robot. For example, if the robot observes

a *reply*$\overline{OK}$ action after activation of *inspect*, it can rule out those situations from its belief state where $\neg FL$ holds. In general, the observation of a *reply* action can be used to *rule out* those situations whose associated pGOLOG model of the low-level processes ll is not about to execute this very *reply* action. To make this precise, we define the predicate *adv&filter*$(s^*, ll^*, do(a, s))$ which - if a is an *reply* action - preserves only those configurations of *advConfig* whose pGOLOG-component is about to execute a.

$$\begin{aligned}
&\textit{adv\&filter}(s^*, ll^*, do(a,s)) = p \equiv \exists s'.s^* = do(a,s') \wedge \\
&\quad [\neg \mathfrak{R}(a) \wedge \textit{advConfig}(s', ll^*, do(a,s)) = p \vee \\
&\quad \mathfrak{R}(a) \wedge [\exists s'', ll'', p'', p^*.\textit{advConfig}(s'', ll'', do(a,s)) = p'' \wedge \\
&\qquad\qquad transPr^*(ll'', s'', ll^*, s^*) = p^* \wedge p'' > 0 \wedge p^* > 0 \wedge p = p'' \cdot p^* \vee \\
&\qquad\qquad p = 0 \wedge \mathfrak{R}(a) \wedge \neg \exists s'', ll''.(\textit{advConfig}(s'', ll'', do(a,s)) > 0 \wedge \\
&\qquad\qquad\quad transPr^*(ll'', s'', ll^*, s^*) > 0)]]
\end{aligned}$$

If a is an ordinary action, *adv&filter* is almost like *advConfig*; the only difference is that all situations s^* considered in $do(a, s)$ now "end" with action a, i.e. $\exists s'.s^* = do(a, s')$. However, if a is a *reply* action, then we keep only those situations s^* in the belief state whose associated pGOLOG model correctly predicted that the *reply* action a would be executed next.

Successor State Axioms for p ***and*** ll**.** It can be shown that the function *adv&filter* is well-defined, meaning that any configuration $\langle s^*, ll^* \rangle$ with positive weight is assigned exactly one weight. Furthermore, it can be shown that for each situation s^* there is at most one ll^* such that *adv&filter*$(s^*, ll^*, do(a, s)) > 0$. Therefore, p and ll can simply be defined as the situation resp. pGOLOG component of *adv&filter*.

$$\begin{aligned}
&p(s^*, do(a,s)) = p \equiv \exists ll^*.\textit{adv\&filter}(s^*, ll^*, do(a,s)) = p \wedge \\
&\quad p > 0 \vee \forall ll^*.\textit{adv\&filter}(s^*, ll^*, do(a,s)) = 0 \wedge p = 0 \\
&ll(s^*, do(a,s)) = ll^* \equiv \textit{adv\&filter}(s^*, ll^*, do(a,s)) > 0 \vee \\
&\quad \forall ll'.\textit{adv\&filter}(s^*, ll', do(a,s)) = 0 \wedge ll^* = nil
\end{aligned}$$

5.3 Examples

To illustrate how p and ll evolve, and in particular how the perception of an exogeneous *reply* action is used to sharpen the robot's beliefs, we will now consider the value of p and ll in different situations. We begin with situation $S_{\text{inspect}} \doteq do([\textit{send}(\textit{fork}, \textit{inspect}), \textit{reply}(\textit{fork}, nil), \textit{tUpdate}(1), ..., \textit{tUpdate}(10)], S_0)$, already mentioned above. Let Γ be the foundational axioms of Section 2 (except for the induction axiom) together with the successor state axioms for p and ll, action precondition axioms stating that all *set* and *clip* actions are always possible, successor state axioms for the fluents *PA*, *FL*, *PR* and *ER*, and the probabilistic characterization of the initial state of Section 4. Then, from Γ we can deduce that in S_{inspect} two situations are considered possible. Intuitively, the first one corresponds to the case where the widget is flawed and the second one to the case where it is not flawed. Furthermore, we can deduce that these situations have an associated pGOLOG model of the low-level processes that accounts for the fact that the *paint* process is active and about to provide a *reply*.

$$\Gamma \models \forall s', ll'.p(s', S_{\text{inspect}}) > 0 \wedge ll(s', S_{\text{inspect}}) = ll' \equiv$$
$$\exists s^*.s' = do([send(fork, inspect), reply(fork, nil), ..., tUpdate(10)], s'_0)$$
$$\wedge [ll' = conc(kernelProc, [start \geq 10?, prob(0.9, reply\overline{OK}, replyOK)]) \vee$$
$$ll' = conc(kernelProc, [start \geq 10?, replyOK])]$$

We remark that so far the robot's belief concerning the value of *FL* remains unchanged ($\Gamma \models Bel(FL, S_{\text{inspect}}) = Bel(FL, S_0)$). Now assume that the inspect process provides a $reply\overline{OK}$ answer, leading to Situation $S_{\neg ok} \doteq do(reply\overline{OK}, S_{\text{inspect}})$. Intuitively, we would expect that after this observation the robot no longer considers a situation possible where the widget is not flawed. Indeed, we can deduce that in $S_{\neg ok}$ the robot only considers one situation possible, and that *FL* holds in this situation.

$$\Gamma \models p(s', S_{\neg ok}) = p \wedge p > 0 \equiv \exists s'_0.\, p(s'_0, S_0) > 0 \wedge$$
$$s' = do([send(fork, inspect), reply(fork, nil), ..., tUpdate(10),$$
$$tossHead, reply\overline{OK}], s'_0) \wedge FL(s') \wedge FL(s'_0)$$

Intuitively, the only situation that remains in the belief state corresponds to the simulation trace where *FL* holds, and *inspect* correctly reports $reply\overline{OK}$. This corresponds to the execution of the first branch of the *prob* instruction in the pGOLOG model *inspectProc*, leading to a *tossHead* action in the resulting execution trace. All other simulation traces would end up in a *replyOK* answer, and are thus ruled out from the belief state by *adv&filter*. We remark that the resulting belief state implies $Bel(FL, S_{\neg ok}) = 1$.

Similarly, if the robot would observe *replyOK*, we could deduce that only two situations are considered possible in the resulting situation $do(replyOK, S_{\text{inspect}})$: one corresponding to the widget beeing flawed (prob. 30%) *and* inspect erroneously reporting *OK* (prob. 10%), and another one where the widget is not flawed (prob. 70%). The robot's resulting belief in *FL* would then correspond to the normalized probability of the first case, which is $(0.3 * 0.1)/(0.7 + 0.3 * 0.1) = \frac{3}{73}$.

As another example, let us consider the situation $S_{uc} \doteq do([send(fork, paint), ..., tUpdate(15)], S_0)$, where the *paint* process is active for 15 seconds. In this situation, the low-level process paint has already caused *UC* to become true, and is waiting until time 30 whereat it may cause *PA* to become true.

$$\Gamma \models p(s', S_{uc}) > 0 \equiv$$
$$\exists s^*.s' = do([send(fork, paint), ..., tUpdate(10), setUC,$$
$$tUpdate(11), ..., tUpdate(15)], s^*) \wedge$$
$$ll(s', S_{uc}) = conc(kernelProc, [start \geq 30?, if(PR, setER, prob(0.95, setPA))])$$

Some seconds later, the robot's belief in *PA* will raise to 95% due to the fact that it will assume that paint has finished execution. However, the robot's belief in the widget beeing flawed will remain unchanged.

6 Belief-Based Programming

As an application of belief update, we will now introduce the concept of *belief-based programs*, GOLOG_{rp} programs that appeal to the robot's beliefs at execution time.[10] In

[10] This is similar to Reiter's notion of knowledge-based programming [Rei00]. However, we remark that here we are dealing with *degrees of belief*.

particular, we introduce a special epistemic test $\mathit{BTest}(\phi, p, s)$, which is true if in situation s the robot's belief in ϕ is p. Formally, $\mathit{BTest}(\phi, p, s)$ is a defined relational fluent which is true iff. $\mathit{Bel}(\phi, s) = p$. Using *BTest* within test conditions, a GOLOG$_{rp}$ plan can appeal to the robot's beliefs *at execution time*. As an example, the following plan specifies that the robot is to activate the *inspect* process until it is sufficiently confident about whether the widget is flawed or not. Thereafter, the widget is painted and processed.[11]

$$
\begin{array}{l}
\mathit{proc}(\mathit{savePaint}, \\
\quad [\ \mathit{while}(\exists p.\mathit{BTest}(\mathit{FL}, p) \wedge p \geq 0.001 \wedge p < 1, \\
\qquad \mathit{send}(\mathit{inspect}, \mathit{nil}), \mathit{forkInspect}, \mathit{reg}(\mathit{inspect})\ /\!\!=\!\!\mathit{nil}?), \\
\quad\ \mathit{forkPaint}, \mathit{waitTime}(30), \\
\quad\ \mathit{if}(\mathit{BTest}(\mathit{FL}, 1), \mathit{forkReject}, \mathit{forkShip}), \mathit{reg}(\mathit{processed})\ /\!\!=\!\!\mathit{nil}?]
\end{array}
$$

We remark that the above program causes at most three activations of *inspect*: as we have seen in the previous section, the observation of *one OK* answer causes the robot's belief in *FL* to drop to $3/73$. Similarly, the observation of three *OK*s causes the robot's belief to drop to $(0.3 * 0.1 * 0.1 * 0.1)/(0.7 + 0.3 * 0.1 * 0.1 * 0.1)$, which is less than 0.001. On the other hand, the observation of a $\overline{\mathit{OK}}$ answer immediately causes the robot's belief in *FL* to rise to 1.

Unlike ordinary GOLOG programs which are conditioned on facts about the world, in belief-based programs like the above actions are conditioned on the robot's belief state at execution time. As the example illustrates, belief-based programs allow the programmer to provide domain dependent procedural knowledge in a natural way. From a pragmatic point of view, belief-based programming can be an attractive alternative to probabilistic planning because it represents a much simpler computational problem. While probabilistic planners are searching for an (optimal) plan from first principles, which in the worst case means that an exponential number of candidate plans has to be projected, the execution of a belief-based program only requires the computation of the belief state of the robot along the execution of a given plan.

Implementation. Just as in the case of ConGolog, it is straightforward to implement a pGOLOG interpreter in PROLOG. We remark that our implementation was able to execute the above belief-based plan in a fraction of a second.

7 Discussion

Summarizing, we have shown how to update the probabilistic belief state of a robot during on-line execution of high-level GOLOG$_{rp}$ plans. To do so, we have modeled a layered robot control architecture within the pGOLOG framework, making use of probabilistic pGOLOG programs to model noisy low-level processes. In order to deal with sensing, we have introduced the concept of sensor processes, low-level processes whose activation results in exogenous *reply* actions. Finally, we have introduced belief-based programs, GOLOG$_{rp}$ programs whose tests appeal to the agent's beliefs at execution time. We remark that unlike approaches like [Lev96,BHL99], we represent the belief state of the agent by a set of possible situations *and an associated model of the state of execution of*

[11] As usual, we leave out the *now* argument in the tests, in particular in the epistemic fluent *BTest*.

the low-level processes, which allows us to account for noisy processes with temporal extent.

The whole framework, in particular the definition of p and ll, relies on the fact that pGOLOG programs are deterministic. As a result, it is not possible to specify unprioritized concurrency as done in ConGolog where the resulting course of actions is not uniquely determined. However, when we consider processes with temporal extent, this does not seem to be a severe restriction, because the priority of a process manifests only when two processes wish to execute an action at exactly the same time; actions with different execution times are not affected.

Probably the closest work to that reported in this paper is that of Bacchus, Halpern and Levesque [BHL99], to which we owe the characterization of the robot's epistemic state. However, while we manage solely with the *prob* instruction to represent noise, they make use of the concepts of *nondeterministic instructions*, *action-likelihood axioms* $OI(a, a', s)$ and *observation-indistinguishability* axioms $l(a, s)$, and represent the execution of noisy actions as atomic. This results in a simpler SSA for p, but at the cost of a more complex specification of the effects of the noisy sensors and effectors. Furthermore, it is not clear how to project a plan within their framework. On the other hand, probabilistic projection in the pGOLOG framework was already considered in [GL00], and it would be relatively straightforward to consider both projection and belief update within pGOLOG.

As for probabilistic planners like C-Buridan [DHW94], they usually completely ignore belief update. Besides, they represent processes as atomic actions. The latter also holds for the theory of POMDPs (which is concerned with both reasoning tasks), but whose computational cost is prohibitive already in relatively small domains. We believe that in many domains, the use of belief-based programs providing procedural knowledge is more promissing than uninformed search for an optimal plan. In [Poo98], Poole proposes an integration of decision theory and the situation calculus, which however is primarily concerned with the expected utility of a candidate plan. Finally, the recently proposed *DTGolog* [BRST00] assumes full observability of the domain. All of these approaches do not account for the temporal extent of the low-level processes.

References

[BCF+00] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2), 2000.

[BHL99] F. Bacchus, J.Y. Halpern, and H. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence 111(1-2)*, 1999.

[BRST00] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI'2000*, 2000.

[dGL99] G. de Giacomo and H.J. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1999.

[DHW94] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. of AIPS'94*, 1994.

[GL00] H. Grosskreutz and G. Lakemeyer. Turning high-level plans into robot programs in uncertain domains. In *ECAI'2000*, 2000.

[GL01] H. Grosskreutz and G. Lakemeyer. Online-execution of ccgolog plans. In *IJCAI*, 2001.

[GLL00] Guiseppe De Giacomo, Yves Lesperance, and Hector J Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.

[KHW95] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.

[KLC98] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence 101(1-2*, 1998.

[Lev96] H. J. Levesque. What is planning in the presence of sensing. In *AAAI'96*, 1996.

[LPR98] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998. URL: `http://www.ep.liu.se/ea/cis/1998/018/`.

[McC63] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University. Reprinted 1968 in Semantic Information Processing, MIT Press, 1963.

[Poo98] David Poole. Decision theory, the situation calculus and conditional plans. *Linköping Electronic Articles in Computer and Information Science*, 3(8), 1998. URL: `http://www.ep.liu.se/ea/cis/1998/008/`.

[Rei00] R. Reiter. On knowledge-based programming with sensing in the situation calculus. In *Second International Cognitive Robotics Workshop*, 2000.

Finding Optimal Solutions to Atomix

Falk Hüffner[1], Stefan Edelkamp[2], Henning Fernau[1], and Rolf Niedermeier[1]

[1] Wilhelm-Schickard Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
{hueffner,fernau,niedermr}@informatik.uni-tuebingen.de
[2] Institut für Informatik, Universität Freiburg,
Georges-Köhler-Allee 51, D-79110 Freiburg, Fed. Rep. of Germany
edelkamp@informatik.uni-freiburg.de

Abstract. We present solutions of benchmark instances to the solitaire computer game Atomix found with different heuristic search methods. The problem is PSPACE-complete. An implementation of the heuristic algorithm A* is presented that needs no priority queue, thereby having very low memory overhead. The limited memory algorithm IDA* is handicapped by the fact that, due to move transpositions, duplicates appear very frequently in the problem space; several schemes of using memory to mitigate this weakness are explored, among those, "partial" schemes which trade memory savings for a small probability of not finding an optimal solution. Even though the underlying search graph is directed, backward search is shown to be viable, since the branching factor can be proven to be the same as for forward search.

1 Introduction

Atomix was invented in 1990 by Günter Krämer and first published by Thalion Software for the popular computer systems of that time. The goal is to assemble a given molecule from atoms (see Fig. 1). The player can select an atom at a time and "push" it towards one of the four directions north, south, west, and east; it will keep on moving until it hits an obstacle or another atom. The game is won when the atoms form the same constellation (the "molecule") as depicted beside the board. A concrete Atomix problem, given by the original atom positions and the goal molecule, is called a *level* of Atomix.

The original game had a time limit and did not count the moves needed; we will instead focus on the analytical aspect and try to minimize the solution length as a goal. Note that we are only interested in *optimal* solutions; in order to just find any solution fast, quite different algorithms would be necessary.

An implementation of this Atomix variation for the X Window System is available as "katomic" from `http://games.kde.org`. A JavaScript version can be played online at `http://www.sect.mce.hw.ac.uk/~peteri/atomix`.

Our solver program written in C++ is able to solve 17 of the 30 problems from the original Atomix and 18 of the 67 problems from katomic optimally. In an appendix, we list a selection of these findings.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 229–243, 2001.

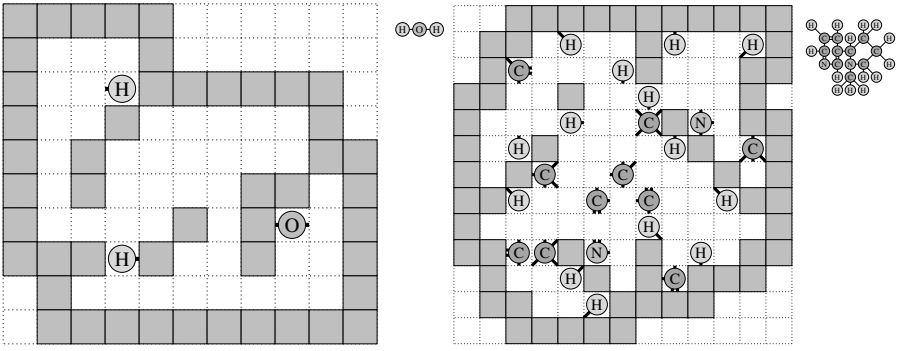


Fig. 1. Two Atomix problems. The left one—which is Atomix 01 in the list of the appendix—can be solved with the following 13 moves, where the atoms are numbered left-to-right in the molecule: 1 down left, 3 left down right up right down left down right, 2 down, 1 right. The right one—which is level number 43 from the "katomic" implementation—illustrates a more complex problem; it takes at least 66 moves to solve.

2 Heuristic Search

Many common problems and, especially, most solitaire puzzles can be formulated as a *state space search* problem: given are a start state, a set of goal states and a set of operators to transform one state into another; wanted is a sequence of operators, also simply called a *move sequence*, that transforms the start state into a goal state and that is of minimal length. A state space can be represented as a graph, with nodes representing states and (directed) edges representing moves. That way, well-known graph algorithms can be applied. To emphasize this aspect, states generated in a state space search are often called "nodes".

For hard combinatorial problems, the use of *heuristics* can often lead to dramatic improvements for a state space search. Many problems would even be unsolvable without them. For a state space search, "heuristic" has a well-defined meaning: an estimate of the moves left from the current state to a goal. Of special interest are *admissible* heuristics: they never overestimate the number of moves. The well-known algorithms A* and IDA* can be proven to always find an optimal solution when using an admissible heuristic. An admissible heuristic judges the "quality" of a state s: if $g(s)$ is the number of moves already applied, and $h(s)$ is the heuristic estimate, then $f(s) := g(s) + h(s)$ is a lower bound on the total number of moves. This number, customarily called the "f-value", can be used in two ways: to guide the search and to reduce the effective depth of the search. The first idea naturally leads to the A* algorithm: "promising" states are examined first. The second is applied in the IDA* algorithm: "hopeless" states are not examined at all.

3 Related Puzzles

The following table compares some search space properties of Atomix to other games. The results are contained in [5,13,15].

Table 1. Search space properties of some puzzles. The effective branching factor is the number of children of a state, after applying memory-bounded pruning methods (in particular, not utilizing transposition tables; see Sect. 5.3 for the methods applied to Atomix). For Sokoban and Atomix, the numbers are for typical puzzles from the human-made test sets; for Sokoban, those problems are about 20×20 and, for Atomix, about 16×16 squares large.

	24-Puzzle	Sokoban	Atomix
Branching factor	2–4	0–50	12–40
effective	2.3	10	7
Solution Length	80–112	97–674	8–120
typical	100	260	45
Search space size	10^{25}	10^{18}	10^{21}
Graph	Undirected	Directed	Directed

Due to its close relationships to Atomix (which will become important in the next section), we discuss the 15- and the 24-puzzle as special instances of the $(n^2 - 1)$-puzzle in more details.

The 15-puzzle consists of a square tray of size 4×4 with 15 tiles numbered 1 through 15 and one empty square. A move consists of sliding one tile adjacent to the empty square into the empty space. The goal is to obtain the usual ordering of the numbers on the tiles by some move sequence. The 15-puzzle is likely to be the most thoroughly analyzed puzzle of this kind [15]. It serves as a kind of "fruit fly" for heuristic search. It is easy to implement, has an obvious heuristic with the "Manhattan distance", and not too large a search space. The Manhattan distance heuristic can be calculated by summing up the number of turns it would take for a tile to get to its goal position if it was the only tile in the tray. This is obviously a lower bound on the actual number of turns.

Many search methods developed for the 15-puzzle can be easily adapted for Atomix. One important difference is that the underlying search graph for Atomix is directed; not every move can be undone.

Improved heuristics for the 15-puzzle make it possible to solve even the extended "24-puzzle"-variation [15]. Most of them follow the common theme of examining a sub-problem where only a few tiles are regarded and most are ignored. The "linear conflict heuristic" [9], for example, tries to find pairs of tiles in a row or column which need to pass each other to get to the goal position. In such a case, another two moves can be added to the heuristic given by the Manhattan distance, since one tile will have to move out of the way and back. The work of Culberson and Schaeffer [2] generalizes this idea to "pattern databases": Each possible distribution of the tiles 1–8 on the board is analyzed and solved, yielding a lower bound which is often better than the Manhattan heuristic with

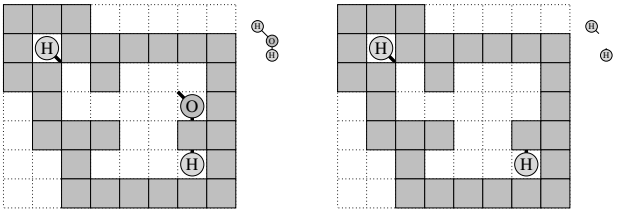


Fig. 2. The left problem can be solved in 13 moves. We cannot get a lower bound by leaving out one atom, as in the right picture; the problem even becomes unsolvable.

the linear conflict heuristic, since there are more tile interactions. The same is done for the other 7 tiles. Unfortunately, these powerful techniques cannot be directly applied to Atomix, since removing atoms from a state does not necessarily make it easier to solve; in fact, it can even become unsolvable, see Fig. 2.

4 Complexity of Atomix

4.1 Complexity of Sliding-Block Puzzles

The time complexity of sliding block puzzles was the subject of intense research in the past. Though seemingly trivial, most variations are at least NP-hard and, some, even PSPACE-complete. The following table shows some results. The table was basically taken from Demaine et al. [3], extended by the category of games where the blocks are pushed by an external agent not represented on the board, into which Atomix falls. The columns mean:

1. Are the moves performed by a robot on the board, or by an outside agent?
2. Can the robot pull as well as push?
3. Does each block occupy a unit square, or may there be larger blocks?
4. Are there fixed blocks, or are all blocks movable?
5. How many blocks can be pushed at a time?
6. Does it suffice to move the robot/a special block to a certain target location, instead of pushing *all* blocks into their goal locations?
7. Will the blocks "keep sliding" when pushed until they hit an obstacle?
8. The dimension of the puzzle: is it 2D or 3D?

Game	1. Robot	2. Pull	3. Blocks	4. Fixed	5. #	6. Path	7. Slide	8. Dim.	9. Complexity
PushPush3D	+	−	unit	−	1	+	+	3D	NP-hard
PushPush	+	−	unit	−	1	+	+	2D	NP-hard
Push-∗	+	−	unit	−	k	−	−	2D	NP-hard
Sokoban+	+	−	1×2	+	2	−	−	2D	PSPACE-compl.
Sokoban	+	−	unit	+	1	−	−	2D	PSPACE-compl. [1]
15-Puzzle	−		unit	−	1	−	−	2D	NP-compl. [17]
Rush Hour	−		1×{2,3}	−	1	+	−	2D	PSPACE-compl.
Atomix	−		unit	+	1	−	+	2D	PSPACE-compl. [12]

4.2 A Formal Definition of Atomix

We will now give a formal definition of an Atomix problem instance (*level*).

Definition 1. *An Atomix problem instance consists of:*

- *A finite set A of so-called* atom types*.*
- *A game* board $B = \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$*.*
- *A bit matrix $O = (O[p] \in \{0,1\} \mid p \in B)$ of size $w \times h$ (the* obstacles*). A* position *is simply a tuple $p = (p_x, p_y) \in B$. A* state *s is defined as a subset of $A \times B$. An element of s is also called an* atom*. Note that the same atom type might appear several times in a state.*
  A position $p = (p_x, p_y)$ is said to be empty *for a state s if $O[p] = 0$ and there is no $a \in A$ with $(a, (p_x, p_y)) \in s$.*
  Positions outside of B areassumed not to be empty.
- *A state S (the* start state*), which satisfies that, for all $(a, p) \in S$, $O[p] = 0$.*
- *A state G (the* goal state*). For the problem to be solvable, for all $(a, p) \in G$, $O[p] = 0$ and there must be a bijection between S and G where each atom in S maps onto an atom in G with the same atom type.*

A direction *(d_x, d_y) is a tuple of x and y offsets, i. e., one of $(0, -1)$, $(1, 0)$, $(0, 1)$ and $(-1, 0)$. A* move *is a tuple of a position p and a direction d. For a state s, a move (p, d) is only legal if there is an atom (a, p) in s, and $(p_x + d_x, p_y + d_y)$ is empty.*

Applying a move (p, d) to a state s will yield another state s' in which every atom has the same position, except the atom (a, p): it will be replaced by (a, p') with $p' = (p_x + \delta d_x, p_y + \delta d_y)$, where $(p_x + \delta' d_x, p_y + \delta' d_y)$ is empty for all $0 < \delta' \leq \delta$, and $(p_x + (\delta + 1)d_x, p_y + (\delta + 1)d_y)$ is not empty. A solution *is a sequence of moves which, incrementally applied to the start state, yields the goal state.*

The main difference between this formal definition and the informal introduction is that the goal positions of the atoms are given explicitly. The reason is that this makes the puzzle both easier to analyze and to implement. Since the number of goal positions is linear in the board size, this difference does not affect the time complexity significantly. Our implementation handles different possible goal positions by imposing a move limit and trying all possible goal positions with that limit, and repeating with an incremented move limit until a solution is found.[1]

4.3 The Hardness of Atomix

Proposition 1. *Atomix on an $n \times n$ board is NP-hard.*

[1] As explained later, this incremental approach is already inherent to IDA*, and can be applied to A* with reasonable overhead.

Proof. Any $(n^2 - 1)$-puzzle instance can be transformed into an Atomix instance by replacing the numbered tiles with atoms of unique atom types. For the $(n^2 - 1)$-puzzle, a legal move consists of sliding a tile into the empty space. In the reduction, those are also the only legal moves, since all atoms not adjacent to the empty square cannot satisfy the move legality condition, and those adjacent to the empty square can only take its place as a move. As shown by Ratner and Warmuth, the $(n^2 - 1)$-puzzle is NP-complete [17], so Atomix is NP-hard. □

Proposition 2. *Atomix on an $n \times n$ board is in PSPACE.*

Proof. A nondeterministic Turing-machine can solve Atomix by repeatedly applying a legal move from the start state encoded on its tape until a goal is reached. The number of possible Atomix states is limited by $n^2!$; hence, the machine can announce that the puzzle is unsolvable after having applied more moves without finding a solution. Since an encoding of an Atomix state needs only polynomial space, it follows that Atomix is in NPSPACE = PSPACE. □

Very recently, Holzer and Schwoon [12] showed by reduction from *non-empty intersection of finite automata* that Atomix is even PSPACE-complete. They also provide a level with an exponentially long optimal solution.

5 Searching the State Space of Atomix

Much progress has been made in the area of heuristic search. This is due to: faster machines with more memory, better heuristics, and better search methods. Of these three, by far, the largest improvements come from better heuristics.

5.1 Heuristics for Atomix

As is often the case, a heuristic for Atomix can be devised by examining a model with relaxed restrictions. We drop the condition that an atom slides as far as possible: it may stop at any closer position. These moves are called *generalized moves*.[2] In order to obtain an easily computable heuristic, we also allow that an atom may also pass through other atoms or share a place with another atom. The goal distance in this model can be summed up for all atoms to yield an admissible heuristic for the original problem.

The following properties are immediate consequences of the definition.

Property 1. The heuristic is admissible.

Property 2. The h-values of child states can only differ from that of the parent state by 0, +1 or −1.

[2] The variant of Atomix which uses generalized moves has an undirected search graph. Atomix with generalized moves on an $n \times n$ board is also NP-hard but is in PSPACE.

Property 3. The heuristic is *monotone* (consistent), i. e., the f-value of a child state cannot be lower than the f-value of the parent state.

Apart from this somewhat obvious heuristic, it proved to be pretty hard to make any improvements. Two ideas were considered, but not implemented due to their limited applicability:

If an atom needs a "stopper" at a certain position to make a turn for each optimal path, but no optimal path of any atom has an intermediate position at the stopper position, h can be incremented by one.

If an atom is alone in a "cave", for some positions, one or two moves can be added to the heuristic (see the example below). A "cave" is an area that contains no goal position and has only one entry; if an atom is alone in there, it cannot use any stoppers unless another atom leaves its optimal path. This heuristic has a greater potential, since it can be added up admissibly for each cave. Unfortunately, only a few levels from our test set contain caves which could yield improved heuristics.

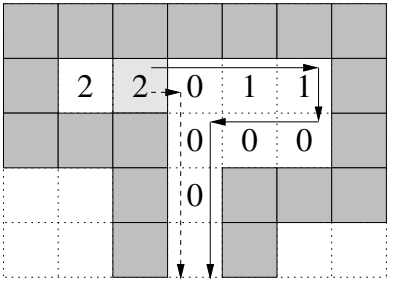


Fig. 3. An example for the "cave"-heuristic: if only one atom is in the cave, the number denoted on its square can be added to the heuristic estimate. For example, an atom on the light grey square has to take the path marked with a solid line, instead of the optimal path of generalized moves marked with a dashed line, which is two moves shorter.

5.2 A*

A* is one of the oldest heuristic search algorithms [10]. It is very time-efficient, but needs an exponential amount of memory. A* remembers all states ever encountered, which is the reason for its exponential space complexity. A priority queue holds all states that have not yet been expanded. It is sorted by the f-value of the states. Nodes are popped from the queue and expanded afterwards. The children are inserted into the queue or discarded if they were already encountered. Sometimes, the same state is reached with a lower g-value; in that case, its entry in the state table has to be updated and it will be re-inserted into the queue. With an admissible heuristic, A* will always find an optimal solution.

The state table is usually implemented as a hash table for fast access and low memory overhead. The priority queue can be implemented with a bucket for each f-value, containing all open states with that f-value. In Sect. 6.2, we present an alternative implementation that only needs the state table and does without a priority queue.

5.3 IDA*

Iterative Deepening A* (IDA*) (see [14]) was the first algorithm that allowed finding optimal solutions to the 15-puzzle. IDA* performs a series of depth-first searches, with an increasing move limit. The heuristic is used to prune subtrees where it is known that the bound will be exceeded, since the f-value is larger than the bound. Each iteration will visit all nodes encountered in the previous iteration again; but, since the majority of nodes will be generated in the last iteration, this does not affect the time complexity.

IDA* uses no memory except for the stack, so its memory use is linear in the search depth. Also, since it needs no intricate data structures, it can be implemented very efficiently. But of course, this comes at a price: IDA* does not detect *transpositions* in the search graph. If a state is encountered that has already been expanded and dismissed, it will be expanded again, possibly resulting in the re-evaluation of a huge subtree. There are two approaches to lessen this weakness: use of problem specific knowledge and use of memory.

Pruning the Search Space. Several techniques are known for pruning, e. g., predecessor elimination, which disallows to take back moves immediately. For games with undirected underlying graphs like the 15-puzzle, this is an obvious optimization. For Atomix, it can still be applied, since pushing an atom into the opposite direction immediately after a move always yields the same state as pushing it in that direction in the first place.

Move Pruning. When examining a solution move sequence for an Atomix level, one notices that many, though not all moves could be interchanged. Interchanging moves is not possible in four cases, as is explained in Fig. 4.

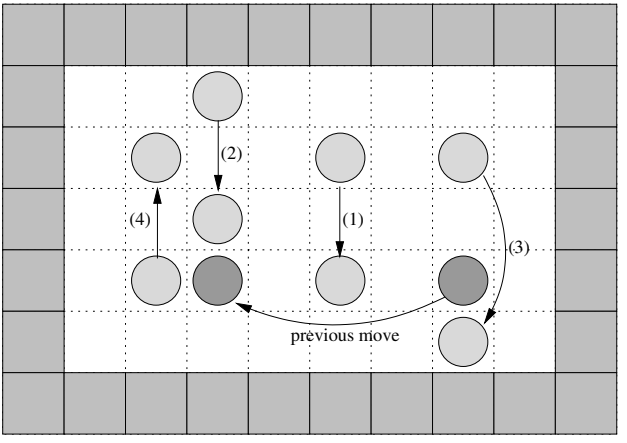


Fig. 4. There are four cases where two moves are dependent, i. e., their order cannot be interchanged: (1) The current atom would have stopped the previously moved atom earlier. (2) The current atom uses the previously moved atom as a stopper. (3) The current atom would stop earlier if the previously moved atom had not been moved. (4) The current atom was the stopper of the previously moved atom.

The idea is to check if a generated move is independent of the previous move (i. e., applying them in reversed order would yield the same state) and,

if they are independent, to impose an arbitrary order (the atom with the lower number must move first). This scheme has proven to be very efficient in avoiding transpositions, reducing running time by several orders of magnitudes.

5.4 Partial IDA*

Analogously to the two-player game search, a *transposition table* can be used to avoid re-expanding states [18]. States are inserted into a hash table together with their g-value as they are generated. Then, for each newly generated state, it is looked up whether it has already been expanded with the same or a lower g-value so it can be pruned. If memory was unlimited, this would avoid all possible transpositions. Many schemes have been proposed for proper management of the transposition table with limited memory [4]; our implementation simply refuses to insert states into an exhausted table.

A lot of memory can be saved with *Partial IDA** [6,7]. This idea originates in the field of *protocol verification*, where the objective is to generate all reachable states and check if they fulfill a certain criterion. A hash table is used to avoid re-expanding states. Just as for a single-agent search, memory is the limiting resource. Therefore, Holzmann suggested *bitstate hashing* [11]: instead of storing the complete state, only a single bit corresponding to the hash value is set, indicating that this state has been visited before. Because of the possibility of hash collisions, states might get pruned erroneously, so this method can give false positives. When applied to IDA*, states on optimal paths could get pruned, so the method looses admissibility, but is still useful to determine upper bounds and likely lower bounds.

For Atomix, initial experiments with Partial IDA* rarely found optimal solutions. The reason is that just knowing a state has been encountered before is not sufficient, because if we encounter it with a lower g-value than previously, it needs to be expanded again. To achieve this, we include g into the hash value and look up with g and $g-1$. This means transpositions with better g will not be found in the table and expanded, as desired. Transpositions with g worse by 2 or more will also not be detected; experiments showed that they are rare and the resulting subtrees are shallow, though.

By probing twice (with g and $g-1$), we increase the likelihood of hash collisions. For example, if we declare the table to be full if every 8th bit is set, we have an effective memory usage of 1 byte per state, and a collision probability of $1-\left(\frac{7}{8}\right)^2 = 23\%$. To improve the collision resistance, one can calculate a second hash value and always set and check two bits, effectively doubling memory usage but lowering collision probability to $1-\left(\frac{63}{64}\right)^2 = 3\%$.

A related scheme with better memory efficiency and collision resistance is *hash compaction* [20]. It utilizes a hash table where, instead of the complete state, only a hash signature is saved. In our implementation, we use 1 byte for the signature, and probe for g and $g-1$. This way, we have a collision probability of $1-\left(\frac{255}{256}\right)^2 = 0.8\%$, so even if there is only a single possible solution of length

30, the probability of finding it is $\left(\left(\frac{255}{256}\right)^2\right)^{30} = 79\%$; and in fact, all 47 solutions found this way were optimal.

Different policies are possible in the case of a hash collision detected by differing signatures. Usual hash table techniques like chaining or open addressing can be applied. We tried a much simpler scheme: the old entry gets overwritten. This can be seen as a special case of the *t-limited scheme* proposed by Stern and Dill [19] with $t = 1$. One disadvantage of this scheme is that entries will already get overwritten before the table is completely full. Since for the "interesting" (difficult) cases, the state table will fill up soon anyway, this effect is limited.

5.5 Backward Search

Many puzzles are *symmetric*, i. e., the set of children of a state equals the set of possible parents. This is equivalent to the state space graph being undirected. As already mentioned, this is the case for the 15-puzzle, but not for Sokoban or Atomix. For Atomix, it is simple to find all potential parent states, though: they can be found by applying all legal *backward moves*. In a backward move, an atom being pushed may stop moving at any position, but it can only be pushed in a direction if it is adjacent to an obstacle in the *opposite* direction.

Formally defined, a backward move is a triple of a position p, a direction d, and a distance δ. It is legal for a state s if there is an atom (a, p) in s, and $(p_x - d_x, p_y - d_y)$ is *not* empty, and $(p_x + \delta' d_x, p_y + \delta' d_y)$ is empty for all $0 < \delta' \leq \delta$. Applying a backward move is analogous to applying a forward move.

Expanding states for backward Atomix is about as easy as for forward Atomix, and the same heuristic can be used, since the generalized moves from Sect. 5.1 comprise backward moves. Hence, the crucial point is the branching factor.

Lemma 1. *The sum of possible forward moves and the sum of possible backward moves of all states of a level are identical and, therefore, the average number of children for backwards expansion is exactly the same as for forward expansion.*

Proof. We first show the equality for a single atom by structural induction. On a board with no empty squares, the equation is trivially true. We show it also remains true when removing an obstacle. The change in the number of moves depends on the pattern of empty squares around the obstacle being removed; we examine all possible patterns (up to symmetry, and omitting the trivial case of 4 obstacles), as illustrated in Fig. 5, with a, b, c and d being the number of empty squares in each direction.

(a) 3 adjacent obstacles: $1 - b + b + 1 = 1 + 1 = 2$.
(b) 2 adjacent obstacles, where the obstacles are diagonally adjacent: $1 - b + b + d + 2 - d + 1 = 1 + 2 + 1 = 4$.
(c) 2 adjacent obstacles, where the obstacles are opposite: $c + 2 - b + 0 - c + b + 2 = 1 + 2 + 1 = 4$.
(d) 1 adjacent obstacle: $c + 2 - b + d + 1 - c + b + 2 - d + 1 = 1 + 3 + 1 + 1 = 6$.

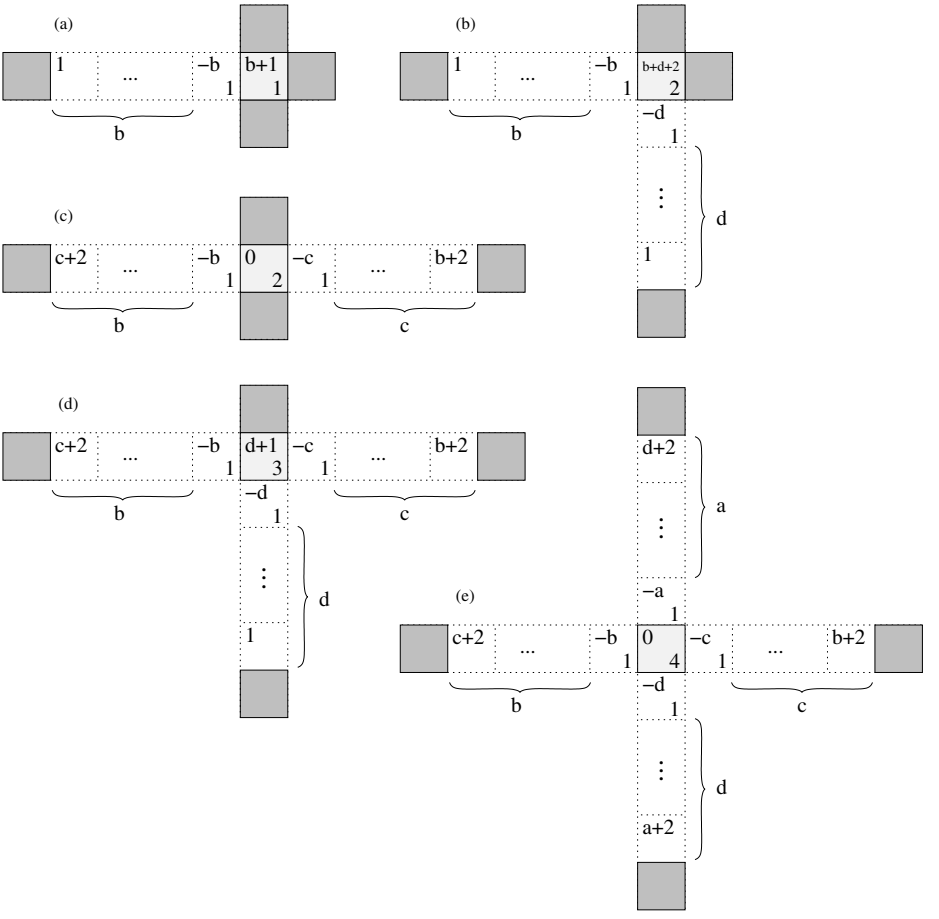


Fig. 5. The light grey obstacle in the center is being removed. The upper left corner of each square denotes the number of backward moves that are lost or gained by this change for an atom on this square. The lower right corner denotes the number of new forward moves. Squares which are skipped in the sketches (denoted by dots) have zero gain with respect to both forward and backward moves.

(e) no adjacent obstacles:
$d + 2 - a + c + 2 - b + 0 - c + b + 2 - d + a + 2 = 1 + 1 + 4 + 1 + 1 = 8.$

Now, let us consider the contribution of one atom to the possible moves. Each possible distribution of the other atoms can be considered as a pattern of obstacles. With the observation just made, the sum of possible forward and backward moves is the same when summing up over all possible positions of the considered atom; so the sum over all possible distributions of the other atoms is also identical and, since this equality holds for each atom, the lemma is true. □

In practice, the branching factors can differ substantially, since the generated states are not random; the move operators make certain states more likely than others, and states close to the goal where (by convention) all atoms are close together are much more likely. In our experiments, we observed differences up to 30% in forward and backward branching factors.

6 Implementation

6.1 Identical Atoms

The presence of undistinguishable atoms (i. e., atoms with identical atom types) poses problems for an implementation: The heuristic cannot simply perform a table lookup to find a lower bound for an atom, since it is not clear which atom should go to which goal position. To find a good lower bound, a *minimum cost perfect matching* has to be done for each set of identical atoms to find the cheapest assignment of atoms to goal positions. Minimum cost perfect matching for a bipartite graph can be solved using minimum cost augmentation in time quadratic in the number of identical atoms [16].

6.2 A*

An implementation of A* needs the following operations: check if a state has been encountered before and with which g-value, find an open state with optimal f-value, mark an open state as closed, and update the g-value of a saved state to a lower value.

This is usually implemented with a hash table and a priority queue which stores all open states. We will show that if the heuristic is monotone, no priority queue is actually needed: an optimal open state can be found efficiently without any additional data structures. Our algorithm is easy to implement and time and space efficient.

Initially, the available memory is allocated for two tables: the *state table* and the *hash table*. As states are generated, they are appended to the end of the state table; states never get deleted. The states are tagged with an *open*-bit and with the g-value. The hash table stores a pointer into the state table at the position corresponding to the hash value of the state; this allows a quick lookup of states. A linear displacement scheme is used to resolve hash collisions. The monotonicity of the heuristic implies that f_{opt}, the currently optimal f-value of an open state, is also monotone over the run of A*. To find an optimal open state, a linear search on the state table is performed until an open state with $f = f_{\text{opt}}$ is found. The following proposition shows that this can be done efficiently:

Proposition 3. *In A* with a monotone heuristic with a hash table and no additional data structure, a state with optimal f-value can be found in amortized time O(branching factor).*

Proof. To achieve this, we need to ensure that, for each f_{opt}-value, when we reach the end of the state table, we have expanded all states with $f = f_{\text{opt}}$, so we don't have to go through the table again. This can be ensured by not upgrading a state in place if it is re-encountered with lower g, but to append it at the end like new states. States with $f < f_{\text{opt}}$ will never be reopenened [8], so this suffices to ensure the desired property.

Two kinds of states will be skipped because their f-value differs from f_{opt}:

- Closed states with $f < f_{\text{opt}}$. We keep a pointer to the very first open state, so only closed states with $f = f_{\text{opt}} - 1$ or $f = f_{\text{opt}} - 2$ have to be skipped; for any branching factor greater than 1, this can be at most twice as many as states with $f = f_{\text{opt}}$ and, with a higher branching factor, their number even becomes negligible.
- Open states with $f > f_{\text{opt}}$. They must have been generated by states with $f = f_{\text{opt}}$ or $f = f_{\text{opt}} - 1$, so their number is linear in the number of states with $f = f_{\text{opt}}$ and the branching factor. □

Our implementation with this scheme is several times faster than a naïve implementation using the C++ STL `priority_queue` and `set`, which are based on heaps, resp., binary trees, with a memory overhead of about 30 bytes per state. On a Pentium III with 500 MHz, it can generate around a million states per second.

A disadvantage of this scheme is that it is not possible to further discriminate among optimal states. A common idea to speed up A* is to sort among states with equal f-values those closer to the top that are further advanced.

To trade time for memory, the A* implementation works iteratively: similarly to IDA*, an artificial upper bound on the number of moves is applied and, if the f-value of a generated state exceeds this bound, it is pruned. If then the search fails, it is restarted with the bound increased by one. This also allows us to take multiple goal positions into account. Due to the exponential behavior, this slows down the search only by a constant factor.

7 Conclusions

Atomix proved itself to be a challenging puzzle; this is corroborated by the recent PSPACE-completeness proof. The classic algorithms A* and IDA* have been implemented and adapted to the problem domain; we have found optimal solutions for many problems from our benchmark set. Our A* implementation with a single data structure for the *open* and *closed* set can solve "smaller" puzzles very efficiently. With Partial IDA* based on hash compaction, we have presented a memory-bounded scheme that makes excellent use of the available memory and has low runtime overhead; improved bounds on the error probability would be useful, though. Further progress is likely to come from improved heuristics rather than from better search methods, since our current heuristic is rather uninformed. We have shown that while the search graph is directed, the backward branching factor does not differ from the forward branching factor; this makes Atomix an interesting testbed for bidirectional algorithms.

References

1. J. C. Culberson. Sokoban is PSPACE-complete. In E. Lodi, L. Pagli, and N. Santoro, editors, *Proc. FUN-98*, pp. 65–76. Carleton Scientific, Waterloo, 1998.
2. J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
3. E. D. Demaine, M. L. Demaine, and J. O'Rourke. PushPush and Push-1 are NP-hard in 2D. In *Proc. 12th Canadian Conf. Computational Geometry*, pp. 211–219, Fredericton, 2000.
4. J. Eckerle and S. Schuierer. Efficient memory-limited graph search. In Proc. *KI-95*, vol. 981 of *LNCS/LNAI*, pp. 101–112. Springer, Berlin, 1995.
5. S. Edelkamp and R. E. Korf. The branching factor of regular search spaces. In *Proc. AAAI-98/IAAI-98*, pp. 299–304. AAAI Press, Menlo Park, 1998.
6. S. Edelkamp, A. L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pp. 75–83, 2001.
7. S. Edelkamp and U. Meyer. Theory and Practice of Time-Space Trade-Offs in Memory Limited Search. This volume.
8. S. Edelkamp and S. Schrödl. Localizing A*. In *Proc. AAAI-00/IAAI-00*, pp. 885–890. AAAI Press, Menlo Park, 2000.
9. O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, 1992.
10. P. E. Hart, N. J. Nilsson and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics*, 4:100–107, 1968.
11. G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proc. International Conf. Protocol Specification, Testing, and Verification*, pp. 339–346, Zürich, 1987. North-Holland, Amsterdam.
12. M. Holzer and S. Schwoon. Assembling Molecules in Atomix is Hard. Technical Report 0101, Institut für Informatik, Technische Universität München, May 2001.
13. A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.
14. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
15. R. E. Korf and L. A. Taylor. Finding optimal solutions to the Twenty-Four Puzzle. In *Proc. AAAI-96/IAAI-96*, pp. 1202–1207. AAAI Press, Menlo Park, 1996.
16. H. W. Kuhn. The Hungarian Method for the Assignment Problem. In *Naval Res. Logist. Quart*, pp. 83–98, 1955.
17. D. Ratner and M. K. Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990.
18. A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.
19. U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, Berichte aus der Informatik, pp. 81–90, 1996. Shaker, Aachen.
20. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. CAV-93*, pp. 59–70. Springer, Berlin, 1993.

A Experimental Results

The experiments were performed on a Pentium III with 500 MHz, utilizing 128 MB of main memory and imposing a time limit of one hour. The source can be found at `http://www-fs.informatik.uni-tuebingen.de/~hueffner`.

Man Best result found by participants of an online game
IDA*-tt IDA* with transposition table
IDA*-r IDA* backward search with transposition table
PIDA* Partial IDA* with hash compaction to 1 byte

Level	Atoms	Goals	Man	A*	IDA*	IDA*-tt	IDA*-r	PIDA*
Atomix 01	3	17		= 13	= 13	= 13	= 13	= 13
Atomix 02	5	6		= 21	= 21	= 21	= 21	= 21
Atomix 03	6	4		= 16	= 16	= 16	= 16	= 16
Atomix 04	6	2		≥ 23	≥ 22	= 23	= 23	= 23
Atomix 05	9	2		≥ 34	≥ 34	≥ 35	≥ 35	≥ 37
Atomix 06	8	4		= 13	= 13	= 13	= 13	= 13
Atomix 07	9	1		≥ 25	≥ 26	= 27	≥ 25	= 27
Atomix 09	7	1		= 20	= 20	= 20	= 20	= 20
Atomix 10	10	2		≥ 28	≥ 28	≥ 28	≥ 27	≥ 30
Atomix 11	5	14		= 14	= 14	= 14	= 14	= 14
Atomix 12	9	4		= 14	= 14	= 14	= 14	= 14
Atomix 13	8	1		= 28	= 28	= 28	= 28	= 28
Atomix 15	12	1		≥ 35	≥ 36	≥ 37	≥ 37	≥ 37
Atomix 16	9	2		≥ 26	≥ 26	≥ 27	≥ 25	≥ 28
Atomix 18	8	4		= 13	= 13	= 13	= 13	= 13
Atomix 22	8	3		≥ 24	≥ 24	≥ 25	≥ 23	≥ 27
Atomix 23	4	20		= 10	= 10	= 10	= 10	= 10
Atomix 26	4	17		= 14	= 14	= 14	= 14	= 14
Atomix 28	10	1		≥ 28	≥ 29	≥ 29	≥ 26	≥ 29
Atomix 29	8	2		= 22	= 22	= 22	= 22	= 22
Atomix 30	8	4		= 13	= 13	= 13	= 13	= 13
Unitopia 01	3	41	11	= 11	= 11	= 11	= 11	= 11
Unitopia 02	4	5	22	= 22	= 22	= 22	= 22	= 22
Unitopia 03	5	12	16	= 16	= 16	= 16	= 16	= 16
Unitopia 04	6	5	20	= 20	= 20	= 20	= 20	= 20
Unitopia 05	6	7	21	= 20	= 20	= 20	= 20	= 20
Unitopia 06	9	2	33	≥ 29	≥ 30	≥ 30	≥ 30	≥ 31
Unitopia 07	10	1	36	≥ 33	≥ 33	≥ 34	≥ 32	≥ 35
Unitopia 08	7	4	25	= 23	= 23	= 23	= 23	= 23
Unitopia 10	8	2	41	≥ 36	≥ 36	≥ 37	≥ 38	≥ 40

Time performance. A* runs out of memory usually much before a runtime of one hour and, so, can establish less stringent bounds. The advantage of using a transposition table for IDA* outweighs its runtime overhead and yields better results in all cases. Reverse search performs similar to forward search, as founded by the theroretical findings. Partial IDA* consistently beats IDA* with conventional hash tables because of better memory utilization and less runtime overhead. Note that most of these differences are expected to be more significant if the time limit is increased.

History-Based Diagnosis Templates in the Framework of the Situation Calculus

Gero Iwan

Aachen University of Technology
Department of Computer Science V
52056 Aachen, Germany
iwan@cs.rwth-aachen.de

Abstract. When agents like mobile robots discover that the world is not as expected after carrying out a sequence of actions, they are interested in what action failures or unnoticed actions could have actually occurred, which would help them rectify the situation. For this purpose, we investigate a kind of *history-based diagnosis* which is appropriate for explaining what went wrong in dynamic domains. It turns out that there are often many diagnoses which are quite similar and differ only in the objects they refer to. In this paper we show how these instances can be compactly represented by introducing so-called *diagnosis templates*. We formalize this approach for an action theory based on the situation calculus and discuss a prototypical implementation of a diagnostic system which generates diagnosis templates according to certain preference criteria.

1 Introduction

Agents who carry out a course of actions inevitably run into the problem that things do not work out as planned. For example, a robot delivering a book may end up losing the book along the way or delivering it to the wrong room. Finding out what went wrong and recovering from it is a difficult problem. In contrast to much traditional work on diagnosis where the focus is on the analysis of "what is wrong", diagnosis in settings like mobile robots acting in a changing environment focuses on "what happened" which we refer to as *history-based diagnosis*.

Given a description of system behavior and, potentially, an (assumed) history of occurred events, the diagnostic task arises from a contradicting observation. Diagnoses that explain observations by conjecturing what happened since an initial situation (i. e., without an explicit history) are, e. g., presented in [10,4]. In [11] so-called *explanatory diagnoses* are studied which are continuations of a given history. It is shown that this kind of diagnosing is analogous to planning.

In our approach to diagnosis we allow adding events not only at the end but at any point of the history. In addition to that we exploit another source of explanation by taking into account the possibility that some history events/actions might not have happened as assumed (or might not have occurred at all). Obviously, in environments with uncertain knowledge about occurrence and outcome of events this kind of reasoning is very important, as is the former one. So both have to be combined, yielding *history-based diagnoses*.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 244–259, 2001.

It turns out that there are often many diagnoses which are quite similar and differ only in the objects they refer to. In this paper we show how these instances can be compactly represented by what we call *history-based diagnosis templates*. Moreover, sometimes there are explanations of the observation that are representable by diagnosis templates but not by diagnoses (cf. Theorem 13 – Remark 14). Furthermore, diagnosis templates can be effectively computed.

History-based diagnosis can be used as (and is intended to be) part of an execution control system of an autonomous agent. In [5] a situation-calculus-based execution monitor is presented which assumes "that all discrepancies between the robot's mental world and reality are the result of exogenous actions, and moreover, that the robot observes all such actions" or at least the effects of these actions. Although this approach is more general than it first appears it still does not take account of the information that can be gained from doubting the (assumed) history. For example, when delivering a book a robot mostly is not able to recover from the sole observation that the book is no longer where it should be whereas the information that the book was lost while carrying it from one room, R1, to another room, R2, or that it was delivered to the wrong room, R3, may help a lot. History-based diagnosis also addresses in an explicit manner the problem that there may be a delay between the occurrence of an action failure or of an unexpected event and the observation of its possibly indirect effect(s). This is not done in [5] nor in implemented systems like SPEEDY [2] and ROGUE [6].

The rest of the paper is organized as follows: In the next section we present an example application which we refer to repeatedly. In the following two sections we address some aspects of the situation calculus used in this paper and present the formalization of our approach to history-based diagnosis. Then the subject of diagnosis preferredness is discussed. Afterwards we introduce diagnosis templates. In the following two sections we outline how most preferable diagnosis templates can be computed. In the final section we give a brief summary and outlook.

2 A Robot Example

As an example, let us consider an autonomous robot whose task it is to bring book B from room R1 into room R2. Suppose the robot is in room R1 already. The robot decides (plans) to carry out the sequence of actions

$$\bar{\eta}^* = [pickup(\mathsf{B}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(\mathsf{B})]$$

and initiates its execution.[1] In the situation obtained after the (assumed) execution of the four actions, $\bar{\eta}^*$ is the (assumed) history and it is derivable that B ought to be in R2. Now the robot receives the message (e. g., by the disappointed would-be recipient) that B is not in R2. This contradicts the assumed history $\bar{\eta}^*$. But what happened actually? Some explanations are:

(1) The robot lost B on its way to R2.
(2) The robot lost its way and entered room R3 instead of R2.

[1] Assume a navigation software as in [3] which usually, but not always leads to successful navigation from one room to another.

(3) The robot failed to grip B during the $pickup$-action.
(4) Somebody took away B after the robot had put it down in R2.

In case (3) a "failure variation" of $pickup$, say $pickup'$, happened instead of the "real" $pickup$-action.

The four explanations correspond to four diagnoses which are modified histories explaining the fact that B is not in R2:

$$\bar{\delta}^{(1)} = [pickup(\mathsf{B}), startfor(\mathsf{R2}), robotloses(\mathsf{B}), arriveat(\mathsf{R2}), putdown']$$

$$\bar{\delta}^{(2)} = [pickup(\mathsf{B}), startfor(\mathsf{R2}), arriveat(\mathsf{R3}), putdown(\mathsf{B})]$$

$$\bar{\delta}^{(3)} = [pickup', startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown']$$

$$\bar{\delta}^{(4)} = [pickup(\mathsf{B}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(\mathsf{B}), somebodytakes(\mathsf{B})]$$

In cases (1) and (3) the "real" $putdown$-action could not have taken place since it is necessary to have an object in order to put it down. Therefore $\bar{\delta}^{(1)}$ and $\bar{\delta}^{(3)}$ contain a $putdown'$-action instead of the $putdown$-action. (Note that only $\bar{\delta}^{(4)}$ is a continuation of $\bar{\eta}^*$.)

Of course, there are many other explanations resp. diagnoses. However, there are explanations that should not be considered a valid diagnosis, e. g.

$$\bar{\alpha} = [pickup(\mathsf{B}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(\mathsf{B}),\\ pickup(\mathsf{B}), startfor(\mathsf{R1}), arriveat(\mathsf{R1}), putdown(\mathsf{B})]$$

The reason why $\bar{\alpha}$ is not a diagnosis (although it is an explanation of the observation) is that the robot would have known if it had brought B back into R1 after bringing it into R2 and therefore it had assumed $\bar{\alpha}$ to be the history instead of $\bar{\eta}^*$. The robot would have known if it has executed (or at least initiated) an action like $pickup$ because such *deliberate actions* happen under the control of the robot unlike *(exogenous) events* like losing a book.

From this simple scenario we can already infer the following requirements: a diagnosis should

- form a possible history (according to the given description of the system behavior);
- explain the observation;
- take into consideration the history assumed so far, i. e., include (in the corresponding order) all history events/actions or *variations* of them;

  This is due to the assumption that an event/action occurs in the history because the agent has good reason to presume something has happened (e. g., there may be uncertainty about the actual effects of actions, but not about their initiation).[2]

  In the robot example, $pickup'$ is a variation of $pickup(\mathsf{B})$, and $arriveat(\mathsf{R3})$ is a variation of $arriveat(\mathsf{R2})$.
- use as additional events only suitable *"insertions"*, i. e., such events that are not under the agent's control but may have occurred and can help to explain the observation.

  In the robot example, $robotloses(\mathsf{B})$ and $somebodytakes(\mathsf{B})$ are such insertions.

Note that in the situation calculus there is no *formal* distinction between actions and events: both are called actions (and formalized equally).

[2] In principle, the non-occurrence of an event/action can be represented by a special dummy event/action as a variation, e. g., by an event/action without any effects.

3 Situation Calculus

To formalize our approach we use the *situation-as-histories variant* [8] of the *situation calculus* [9] and assume the reader to be somewhat familiar with it. Nevertheless most of the material presented here can be understood intuitively. In this section we only mention some aspects of the general situation calculus but describe extensions that we use in the context of history-based diagnosis. We adopt the convention that free variables are implicitly universally quantified unless otherwise stated. $\Phi\lceil x_1, \ldots, x_n\rceil$ indicates that the free variables of the formula Φ are among $x_1, \ldots, x_n$.

A basic action theory $\mathcal{D}$ describes the initial state of the world and how the world evolves under the effects of actions. Among others, it consists of successor state axioms [13] and action precondition axioms. For instance, if A is a n-ary action function, an action precondition axiom of the form

$$Poss(A(x_1, \ldots, x_n), s) \equiv \Pi_A\lceil x_1, \ldots, x_n, s\rceil$$

is used to state under which condition, Π_A, it is possible to execute action $A(x_1, \ldots, x_n)$ in situation s, e. g.,

$$Poss(putdown(x), s) \equiv Having(x, s) \wedge \neg Moving(s)$$

Here, in the context of history-based diagnosis, the basic action theory $\mathcal{D}$ additionally contains

- an action variation axiom for each action function
- an insertion axiom

which use predicate symbols $Varia$ and $Inser$ that are conceptually similar to $Poss$.

Action variation axioms of the form

$$Varia(a, A(x_1, \ldots, x_n), s) \equiv \Theta_A\lceil a, x_1, \ldots, x_n, s\rceil$$

are used to state under which condition, Θ_A, an action, a, is a valid variation of another action, $A(x_1, \ldots, x_n)$, in a situation, s, e. g.,

$$Varia(a, putdown(x), s) \equiv a = putdown' \vee \exists y\,[a = putdown(y) \wedge Having(y, s)]$$
$$Varia(a, arriveat(r), s) \equiv \exists r'\,[a = arriveat(r') \wedge Room(r')]$$

Mostly, as in the examples given above, $Poss(a, s)$ implies $Varia(a, a, s)$, i. e., an action is a variation of itself if it is possible to execute it.[3]

An insertion axiom of the form

$$Inser(a, s) \equiv \Theta\lceil a, s\rceil$$

states under which condition, Θ, an action, a, is a valid insertion in a situation, s, e. g.,

$$\begin{aligned} Inser(a, s) \equiv \quad & \exists z\,[a = robotloses(z) \wedge Having(z, s)] \\ \vee\ & \exists z\,[a = somebodytakes(z) \wedge Portable(z)] \end{aligned}$$

Insertions are suitable additional actions that may have occurred and can assist in diagnosing. Typically, insertions are actions (events) that are not under the agent's control.

[3] Sometimes there are good reasons for variation relations without this property. But we do not consider such applications here.

To simplify matters, we assume that the variations and insertions are disjoint, i. e., $\mathcal{D} \models \neg\exists a,s\,[\exists a'\ \mathit{Varia}(a,a',s) \wedge \mathit{Inser}(a,s)]$.

There is a syntactic restriction on the formulas $\Pi_A\lceil\dots,s\rceil, \Theta_A\lceil\dots,s\rceil, \Theta\lceil\dots,s\rceil$: they are uniform in s, i. e., they are formulas such that, if they contain a situation term at all, then that term is s and there is no quantification over s.

In what follows we also need the notion of situation-suppressed formulas, i. e., formulas where all occurrences of situation terms are "deleted" (details omitted). If ϕ is a situation-suppressed formula then $\phi\lfloor\sigma\rfloor$ denotes the situation calculus formula obtained after restoring suppressed situation arguments by "inserting" the situation σ where necessary, e. g., $\phi = \neg\mathit{Moving} \wedge \exists x\,(\mathit{Book}(x) \wedge \mathit{Location}(x,\mathsf{R}))$ is a situation-suppressed formula and $\phi\lfloor\sigma\rfloor = \neg\mathit{Moving}(\sigma) \wedge \exists x\,(\mathit{Book}(x) \wedge \mathit{Location}(x,\mathsf{R},\sigma))$.

We denote action sequences in square brackets, like $[\alpha_1,\dots,\alpha_n]$. $\mathit{do}(\bar\alpha,\sigma)$ denotes the situation obtained after the execution of action sequence $\bar\alpha$ in situation σ.[4] The constant S_0 denotes the initial situation. We abbreviate $\phi\lfloor\mathit{do}(\bar\alpha,S_0)\rfloor$ by $\phi\lfloor\bar\alpha\rfloor$ for every situation-suppressed formula ϕ and use the abbreviation

$$\mathit{Exec}([\alpha_1,\dots,\alpha_n]) \quad \doteq \quad \textstyle\bigwedge_{j\in\{1,\dots,n\}} \mathit{Poss}(\alpha_j, \mathit{do}([\alpha_1,\dots,\alpha_{j-1}],S_0))$$

$\mathit{Exec}([\alpha_1,\dots,\alpha_n])$ expresses that starting in the initial situation S_0 it is possible to execute the actions $\alpha_1,\dots,\alpha_n$ one after another. A ground action sequence $\bar\alpha$ is called *executable* iff $\mathcal{D} \models \mathit{Exec}(\bar\alpha)$. Finally, $\bar\alpha{\cdot}\alpha = [\alpha_1,\dots,\alpha_n,\alpha]$ when $\bar\alpha = [\alpha_1,\dots,\alpha_n]$.

4 History-Based Diagnoses

A *history* is simply a ground action sequence $\bar\eta$. An *observation* is a situation-suppressed closed formula ϕ. The diagnostic task arises if the observation contradicts the assumed history, i. e., $\mathcal{D} \models \neg\phi\lfloor\bar\eta\rfloor$. The (assumed) history and observation of the robot example are

$$\bar\eta^* = [\mathit{pickup}(\mathsf{B}), \mathit{startfor}(\mathsf{R2}), \mathit{arriveat}(\mathsf{R2}), \mathit{putdown}(\mathsf{B})]$$
$$\phi^* = \neg\mathit{Location}(\mathsf{B},\mathsf{R2})$$

Here $\mathcal{D} \models \mathit{Location}(\mathsf{B},\mathsf{R2},\mathit{do}(\bar\eta^*,S_0))$, therefore $\mathcal{D} \models \neg\phi^*\lfloor\bar\eta^*\rfloor$, whereas $\mathcal{D} \models \phi^*\lfloor\bar\eta\rfloor$ should hold for the "real" history $\bar\eta$.

Next we address the rather syntactic qualities of history-based diagnoses.

Definition 1. An *extended variation* of a ground action sequence $\bar\alpha = [\alpha_1,\dots,\alpha_n]$ is a ground action sequence $\bar\delta = [\delta_1,\dots,\delta_m]$ such that

- a mapping $\iota : \{1,\dots,n\} \to \{1,\dots,m\}$ exists with $\iota(1) < \dots < \iota(n)$
- for each $i \in \{\iota(1),\dots,\iota(n)\}$ with $i = \iota(j)$:
  δ_i is a valid variation of α_j in the situation after the action sequence $[\delta_1,\dots,\delta_{i-1}]$, i. e., $\mathcal{D} \models \mathit{Varia}(\delta_i,\alpha_j,\mathit{do}([\delta_1,\dots,\delta_{i-1}],S_0))$
- for each $i \in \{1,\dots,m\} \setminus \{\iota(1),\dots,\iota(n)\}$:
  δ_i is a valid insertion in the situation after the action sequence $[\delta_1,\dots,\delta_{i-1}]$, i. e., $\mathcal{D} \models \mathit{Inser}(\delta_i,\mathit{do}([\delta_1,\dots,\delta_{i-1}],S_0))$ ▪

[4] $\mathit{do}([\alpha_1,\dots,\alpha_n],\sigma)$ abbreviates $\mathit{do}(\alpha_n,\dots\mathit{do}(\alpha_1,\sigma)\dots)$.

The property of $\bar{\delta}$ being an extended variation of $\bar{\alpha}$ can be formulated as a situation calculus formula (Lemma 2). For this purpose we introduce an abbreviation $ExtVari(\bar{\delta}, \bar{\alpha})$ by

$$\begin{aligned}
ExtVari([\,],[\,]) &\doteq True\\
ExtVari([\,],\bar{\alpha}.\alpha) &\doteq False\\
ExtVari(\bar{\delta}.\delta,[\,]) &\doteq Inser(\delta, do(\bar{\delta}, S_0)) \wedge ExtVari(\bar{\delta},[\,])\\
ExtVari(\bar{\delta}.\delta,\bar{\alpha}.\alpha) &\doteq [Varia(\delta,\alpha,do(\bar{\delta},S_0)) \wedge ExtVari(\bar{\delta},\bar{\alpha})]\\
&\quad \vee [Inser(\delta,do(\bar{\delta},S_0)) \wedge ExtVari(\bar{\delta},\bar{\alpha}.\alpha)]
\end{aligned}$$

The first and second equation together mean that $[\,]$ is an extended variation of no other action sequence but $[\,]$. The third equation means that $\bar{\delta}.\delta$ is an extended variation of $[\,]$ iff δ is a valid insertion in the situation $do(\bar{\delta}, S_0)$ and $\bar{\delta}$ is an extended variation of $[\,]$ itself. The fourth equation means that $\bar{\delta}.\delta$ is an extended variation of $\bar{\alpha}.\alpha$ iff *(1.)* δ is a valid variation of α in the situation $do(\bar{\delta}, S_0)$ and $\bar{\delta}$ is an extended variation of $\bar{\alpha}$ or *(2.)* δ is a valid insertion in the situation $do(\bar{\delta}, S_0)$ and $\bar{\delta}$ is an extended variation of $\bar{\alpha}.\alpha$.

Lemma 2. A ground action sequence $\bar{\delta}$ is an extended variation of a ground action sequence $\bar{\alpha}$ iff $\mathcal{D} \models ExtVari(\bar{\delta}, \bar{\alpha})$. ∎

Now it is easy to define history-based diagnoses.

Definition 3. An *explanatory history-based diagnosis* for an observation ϕ and a history $\bar{\eta}$ is an extended variation $\bar{\delta}$ of $\bar{\eta}$ such that $\bar{\delta}$ is executable and ϕ holds in the situation after the action sequence $\bar{\delta}$, i. e., $\mathcal{D} \models \phi[do(\bar{\delta}, S_0)]$. ∎

Definition 3 captures all of the above-mentioned requirements and can also be formulated as a situation calculus formula (Lemma 4). We introduce the abbreviation

$$ExplDiag(\bar{\delta}, \phi, \bar{\eta}) \doteq ExtVari(\bar{\delta}, \bar{\eta}) \wedge Exec(\bar{\delta}) \wedge \phi[\bar{\delta}]$$

Lemma 4. A ground action sequence $\bar{\delta}$ is an explanatory history-based diagnosis for an observation ϕ and a history $\bar{\eta}$ iff $\mathcal{D} \models ExplDiag(\bar{\delta}, \phi, \bar{\eta})$. ∎

In the robot example $\bar{\delta}^{(1)}, \ldots, \bar{\delta}^{(4)}$ are explanatory history-based diagnosis for observation ϕ^* and history $\bar{\eta}^*$. Other simple diagnoses are for instance

$$\begin{aligned}
\bar{\delta}^{(5)} &= [pickup(\mathsf{A}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(\mathsf{A})]\\
\bar{\delta}^{(6)} &= [pickup(\mathsf{B}), startfor(\mathsf{R2}), robotloses(\mathsf{B}), arriveat(\mathsf{R3}), putdown']\\
\bar{\delta}^{(7)} &= [pickup(\mathsf{A}), startfor(\mathsf{R2}), arriveat(\mathsf{R3}), putdown']
\end{aligned}$$

but $\bar{\delta}^{(6)}$ should not be considered a preferred diagnosis since it combines $\bar{\delta}^{(1)}$ and $\bar{\delta}^{(2)}$ and hence "over-explains" ϕ^* in a way. The same is true for $\bar{\delta}^{(7)}$.

5 Preference Values

In most cases there are many diagnoses for a given observation and history (in fact, infinitely many). Therefore criteria are needed to decide which diagnoses are preferred. The number of modifications between a diagnosis and the history, i. e., the number of insertions and proper variations in a diagnosis, yields a simple preference criterion: the higher that number is the less preferred a diagnosis is. When comparing

$$\bar{\delta}^{(8)} = [pickup(\mathsf{B}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown']$$
$$\bar{\delta}^{(2)} = [pickup(\mathsf{B}), startfor(\mathsf{R2}), arriveat(\mathsf{R3}), putdown(\mathsf{B})]$$
$$\bar{\delta}^{(3)} = [pickup', startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown']$$

a problem with this number-of-modifications criterion can be seen: There is one modification in $\bar{\delta}^{(2)}$; in $\bar{\delta}^{(3)}$ there are two, but the $putdown$-failure is merely an aftereffect of the $pickup$-failure and hence should not be counted.

A somewhat more sophisticated preference criterion (which seems to avoid this problem) is comparing diagnoses by inspecting the modifications (not only counting them): a diagnosis is more preferred than another diagnosis if its modifications are a proper subset of the modifications of the other diagnosis. However, this set-of-modifications criterion does not allow to compare arbitrary diagnoses but only those whose modification sets are ordered by the subset relation. Furthermore, according to this set-of-modifications criterion $\bar{\delta}^{(8)}$ is more preferred than $\bar{\delta}^{(3)}$. But it may be the case that an initial $pickup$-failure (which entails a $putdown$-failure) is more likely than an isolated $putdown$-failure and therefore $\bar{\delta}^{(3)}$ should be more preferred than $\bar{\delta}^{(8)}$.

So we look for preference criteria that can subsume the set-of-modifications criterion but also can regard entailed actions and allow to compare all diagnoses with a more subtle ranking than the number-of-modifications criterion. For this purpose we use *preference valuations* $\wp_{\phi,\bar{\eta}}$ which are functions that assign a preference value $\wp_{\phi,\bar{\eta}}(\bar{\delta}) \geq 0$ to every extended variation $\bar{\delta}$ of the history $\bar{\eta}$ (hence to every diagnosis, too) such that higher values for diagnoses correspond to more preferred diagnoses.[5] Particularly we exploit the special structure of *normalized inert modular* preference valuations which are defined at first and explained after that. The idea behind modularity is that the preference value of a diagnosis can be determined using kind of preference values for the single diagnosis actions.

Definition 5. $\wp_{\phi,\bar{\eta}}$ is *normalized* iff $\wp_{\phi,\bar{\eta}}(\bar{\eta}) = 1$. ▪

Definition 6. $\wp_{\phi,\bar{\eta}}$ is *modular* iff for each pair $\langle \delta, \bar{\delta}' \rangle$ where $\bar{\delta}'.\delta$ is a first part of an extended variation of $\bar{\eta}$ there is a value $\wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ such that $0 \leq \wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}') \leq 1$ and

$$\wp_{\phi,\bar{\eta}}(\bar{\delta}) = \wp_{\phi,\bar{\eta}}(\bar{\eta}) \cdot \textstyle\prod_{i \in \{1,\ldots,m\}} \wp_{\phi,\bar{\eta}}(\delta_i \,|\, [\delta_1, \ldots, \delta_{i-1}]) \qquad (*)$$

for every extended variation $\bar{\delta} = [\delta_1, \ldots, \delta_m]$ of $\bar{\eta}$. ▪

[5] Note that, in the majority of cases, for a given preference criterion one can find a corresponding preference valuation subsuming the preference criterion.

Corollary 7. If $\wp_{\phi,\bar{\eta}}$ is modular then $\wp_{\phi,\bar{\eta}}(\eta_{j+1} \,|\, [\eta_1, \ldots, \eta_j]) = 1$. ▪

Definition 8. $\wp_{\phi,\bar{\eta}}$ is *inert modular* iff $\wp_{\phi,\bar{\eta}}$ is modular and $\wp_{\phi,\bar{\eta}}(\eta_{j+1} \,|\, \bar{\delta}') = 1$ when η_{j+1} is the next history action that is not "covered" by $\bar{\delta}'$ (i. e.: $\eta_1, \ldots, \eta_j$ are the history actions for which variations are in $\bar{\delta}'$). ▪

Theorem 9. If a preference valuation respects the set-of-modifications criterion then it is inert modular. ▪

Let us now have a look at the role of the factors $\wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ in Equation $(*)$. Since $0 \leq \wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}') \leq 1$ this factor specifies how the modification associated with δ (i. e., the replacing of a history action by δ or the inserting of δ) diminishes the preference value of the obtained action sequence w. r. t. the preference value $\wp_{\phi,\bar{\eta}}(\bar{\eta})$ of the history. The intuitive meaning of $\wp_{\phi,\bar{\eta}}(\delta_i \,|\, [\delta_1, \ldots, \delta_{i-1}])$ is:

- If δ_i is a variation of the next history action η_{j+1} that is not "covered" by $\delta_1, \ldots, \delta_{i-1}$ (i. e.: $\eta_1, \ldots, \eta_j$ are the history actions for which variations are among $\delta_1, \ldots, \delta_{i-1}$) then $\wp_{\phi,\bar{\eta}}(\delta_i \,|\, [\delta_1, \ldots, \delta_{i-1}])$ is the diminishing factor corresponding to next replacing η_{j+1} by δ_i when the first part of a diagnosis is $[\delta_1, \ldots, \delta_{i-1}]$.

  If $\wp_{\phi,\bar{\eta}}$ is inert then $\wp_{\phi,\bar{\eta}}(\delta_i \,|\, [\delta_1, \ldots, \delta_{i-1}]) = 1$ when $\delta_i = \eta_{j+1}$, i. e.: not-replacing a history action does not diminish the preference value.
- If δ_i is an insertion then $\wp_{\phi,\bar{\eta}}(\delta_i \,|\, [\delta_1, \ldots, \delta_{i-1}])$ is the diminishing factor corresponding to next inserting δ_i when the first part of a diagnosis is $[\delta_1, \ldots, \delta_{i-1}]$.

In both cases we may have $\wp_{\phi,\bar{\eta}}(\delta_i \,|\, [\delta_1, \ldots, \delta_{i-1}]) = 1$ if $[\delta_1, \ldots, \delta_{i-1}]$ entails δ_i.

If $\wp_{\phi,\bar{\eta}}$ is modular then $0 \leq \wp_{\phi,\bar{\eta}}(\bar{\delta}) \leq \wp_{\phi,\bar{\eta}}(\bar{\eta})$ for every extended variation $\bar{\delta}$ of the assumed history $\bar{\eta}$, i. e., $\bar{\eta}$ has the maximal preference value. This is intuitively right: Assuming that the observation does not contradict the history then the history itself is a diagnosis and, of course, the most preferred one. $\wp_{\phi,\bar{\eta}}(\bar{\eta})$ should not equal 0 because otherwise all preference values are 0. If $\wp_{\phi,\bar{\eta}}(\bar{\eta}) \not= 0$then $\wp_{\phi,\bar{\eta}}$ can be normalized. If $\wp_{\phi,\bar{\eta}}$ is normalized then $0 \leq \wp_{\phi,\bar{\eta}}(\bar{\delta}) \leq 1$ and

$$\wp_{\phi,\bar{\eta}}([\delta_1, \ldots, \delta_m]) = \prod_{i \in \{1,\ldots,m\}} \wp_{\phi,\bar{\eta}}(\delta_i \,|\, [\delta_1, \ldots, \delta_{i-1}])$$

So $\wp_{\phi,\bar{\eta}}(\bar{\delta})$ and $\wp_{\phi,\bar{\eta}}(\delta_i \,|\, [\delta_1, \ldots, \delta_{i-1}])$ have the flavor of (conditional) probabilities. But they are no probabilities! For instance, $\sum_{\bar{\delta}} \wp_{\phi,\bar{\eta}}(\bar{\delta}) = 1$ does not hold in general. If every $\sum_{\delta} \wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ is finite one can transform the preference values to probabilities by normalizing them. But then $\wp_{\phi,\bar{\eta}}(\bar{\eta}) < 1$. In [7] an approach is shown how probabilities can be utilized as preference criterion.[6]

Without the requirements of probability distributions we are free to assign intuitive values between 0 and 1 (0 and 1 included) to $\wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ with the only restriction that $\wp_{\phi,\bar{\eta}}(\eta \,|\, \bar{\delta}') = 1$ must hold for the next history action η (see above). In doing so we get an inert modular preference valuation $\wp_{\phi,\bar{\eta}}$ by means of Equation $(*)$.[7] The preference

[6] The "features" that are used in [7] to reduce the complexity of determining the probabilities are comparable with the value-conditions χ that are used later in this paper.

[7] The number-of-modifications criterion (which, of course, subsumes the set-of-modifications criterion) can be emulated by assigning a fixed value other than 0 or 1 to all $\wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ that do not have to be 1.

values provide a preference criterion that can subsume the set-of-modifications criterion but also can regard entailed actions and allows to compare all diagnoses with a possibly more subtle ranking than the number-of-modifications criterion. Note that we only have to provide values for $\wp_{\phi,\bar{\eta}}(\delta \mid \bar{\delta}')$ if δ is a valid variation or insertion in situation $do(\bar{\delta}', S_0)$.

6 Diagnosis Templates

Recall the history and observation of the robot example:

$$\bar{\eta}^* = [pickup(\mathsf{B}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(\mathsf{B})]$$
$$\phi^* = \neg Location(\mathsf{B}, \mathsf{R2})$$

For instance, there may be many diagnoses that arise from erroneously picking up the wrong book in the beginning:

$$[pickup(\mathsf{C}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(\mathsf{C})]$$
$$[pickup(\mathsf{D}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(\mathsf{D})]$$
$$[pickup(\mathsf{E}), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(\mathsf{E})]$$
$$\ldots$$

They all are instances of the non-ground action sequence

$$\bar{\delta} = [pickup(x), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(x)]$$

But not all instances of $\bar{\delta}$ are diagnoses, e. g., with $\{x \mapsto \mathsf{B}\}$, most of them are not even executable or extended variations of the history, e. g., with $\{x \mapsto \mathsf{R1}\}$. In order to further restrict the (ground) instances of (possibly non-ground) action sequences we use parameter constraints.

Definition 10. A *parameter constraint* for an action sequence $\bar{\delta}$ is a formula Ψ such that the free variables of Ψ are exactly the variables in $\bar{\delta}$. ▪

We call $\bar{\delta}'$ an *instance of* $\langle \bar{\delta}, \Psi \rangle$ iff an assignment ν of the variables in $\bar{\delta}$ to terms exists such that $\mathcal{D} \models \overline{\exists} \Psi\nu$ and $\bar{\delta}' = \bar{\delta}\nu$. If $\bar{\delta}\nu$ is ground then $\mathcal{D} \models \Psi\nu$ must hold.

Definition 11. An *explanatory history-based diagnosis template* for an observation ϕ and a history $\bar{\eta}$ is a pair $\langle \bar{\delta}, \Psi \rangle$ consisting of an action sequence $\bar{\delta}$ and a parameter constraint Ψ for $\bar{\delta}$ such that

$$\mathcal{D} \models \overline{\forall}[\Psi \supset ExtVari(\bar{\delta}, \bar{\eta})] \wedge \overline{\exists}[\Psi \wedge Exec(\bar{\delta}) \wedge \phi[\bar{\delta}]]$$

$\langle \bar{\delta}, \Psi \rangle$ is called *universal* iff $\mathcal{D} \models \overline{\forall}[\Psi \wedge Exec(\bar{\delta}) \supset \phi[\bar{\delta}]]$.

$\langle \bar{\delta}, \Psi \rangle$ is called *executable* iff $\mathcal{D} \models \overline{\forall}[\Psi \supset Exec(\bar{\delta})]$. ▪

For example, if the history and observation are

$$\bar{\eta} = [pickup(\mathsf{A}), pickup(\mathsf{B}), startfor(\mathsf{R2}), arriveat(\mathsf{R2})]$$
$$\phi = \neg Location(\mathsf{B}, \mathsf{R2}) \wedge \exists x\, Having(x)$$

then with $\sigma = do([pickup(\mathsf{A}), pickup(\mathsf{B}), startfor(\mathsf{R2})], S_0)$ both $\langle \bar{\delta}', \Psi' \rangle$ where

$\bar{\delta}' = [pickup(\mathsf{A}), pickup(\mathsf{B}), startfor(\mathsf{R2}), robotloses(x), robotloses(y), arriveat(\mathsf{R2})]$
$\Psi' = Having(x, \sigma) \wedge Having(y, do([robotloses(x)], \sigma))$

and $\langle \bar{\delta}'', \Psi'' \rangle$ where

$\bar{\delta}'' = [pickup(\mathsf{A}), pickup(\mathsf{B}), startfor(\mathsf{R2}), robotloses(x), arriveat(\mathsf{R2}), robotloses(y)]$
$\Psi'' = Having(x, \sigma) \wedge Having(y, do([robotloses(x), arriveat(\mathsf{R2})], \sigma))$

are executable diagnosis templates.[8] The first one is universal and all its ground instances, one with $\{x \mid \neg\mathsf{A}, y \mid \neg\mathsf{B}\}$ and one with $\{x \mid \neg\mathsf{B}, y \mid \neg\mathsf{A}\}$, are diagnoses. The second one is not universal since, because of ϕ, its instance with $\{x \mid \neg\mathsf{A}, y \mid \neg\mathsf{B}\}$ is not a diagnosis while its instance with $\{x \mid \neg\mathsf{B}, y \mid \neg\mathsf{A}\}$ is.

From Definition 11 (resp. Corollary 12) one can see: (*1.*) all ground instances of a diagnosis template are extended variations of the history; (*2.*) if it is universal then all executable ground instances are diagnoses; (*3.*) if it is universal and executable then all its ground instances are diagnoses. But it is not necessarily true that at least one ground instance exists that is a diagnosis (see below). This is an advantageous feature of diagnosis templates (cf. Remark 14).

Corollary 12. Let $\langle \bar{\delta}, \Psi \rangle$ be an explanatory history-based diagnosis template for an observation ϕ and a history $\bar{\eta}$. Then $\mathcal{D} \models \overline{\exists}[\Psi \wedge ExplDiag(\bar{\delta}, \phi, \bar{\eta})]$.
If $\langle \bar{\delta}, \Psi \rangle$ is universal then $\mathcal{D} \models \overline{\forall}[\Psi \wedge Exec(\bar{\delta}) \supset ExplDiag(\bar{\delta}, \phi, \bar{\eta})]$.
If $\langle \bar{\delta}, \Psi \rangle$ is universal and executable then $\mathcal{D} \models \overline{\forall}[\Psi \supset ExplDiag(\bar{\delta}, \phi, \bar{\eta})]$. ∎

To every diagnosis template $\langle \bar{\delta}, \Psi \rangle$ the set $\mathrm{D}_{\phi,\bar{\eta}}\langle \bar{\delta}, \Psi \rangle$ of all its ground instances that are diagnoses is assigned. Let $\mathrm{D}_{\phi,\bar{\eta}}$ be the union of all the $\mathrm{D}_{\phi,\bar{\eta}}\langle \bar{\delta}, \Psi \rangle$. Note that for every diagnosis $\bar{\delta}$ $\langle \bar{\delta}, True \rangle$ is a diagnosis template with $\mathrm{D}_{\phi,\bar{\eta}}\langle \bar{\delta}, True \rangle = \{\bar{\delta}\}$. Hence:

Theorem 13. $\mathrm{D}_{\phi,\bar{\eta}}$ is the set of all diagnoses for observation ϕ and history $\bar{\eta}$,
i. e., we do not miss any diagnosis when considering diagnosis templates only.

We do not only not miss any diagnosis but sometimes we actually gain evidence for "diagnostic explanations" that are not expressible by a diagnosis but are representable by a diagnosis template. For instance, assume that in the robot example the robot has the information that in the beginning there are books in room R1 beside B but no further information is given about these books (particularly, the robot does not know which books). Then $\langle \bar{\delta}, \Psi \rangle$ where

$$\bar{\delta} = [pickup(x), startfor(\mathsf{R2}), arriveat(\mathsf{R2}), putdown(x)]$$
$$\Psi = Book(x) \wedge x \not= \mathsf{B} \wedge Location(x, \mathsf{R1}, S_0)$$

is a diagnosis template but none of its ground instances is a diagnosis. In fact, it is an universal and executable diagnosis template such that no ground instance exists at all (since no ground term t exists such that $\mathcal{D} \models \Psi\{x \mid \neg t\}$).

Remark 14. By means of diagnosis templates it is possible to find "diagnoses" (meant in an informal way) that cannot be expressed as diagnoses (as defined by Definition 3). ∎

[8] Suppose that in S_0 the robot does not carry anything and is located in the same room as the books A and B.

Of course, $\langle \bar{\delta}, \mathit{ExplDiag}(\bar{\delta}, \phi, \bar{\eta})\rangle$ is an executable universal diagnosis template iff $\mathcal{D} \models \overline{\exists}\mathit{ExplDiag}(\bar{\delta}, \phi, \bar{\eta})$. The problem with this diagnosis template is that nevertheless one has to compute the action sequence $\bar{\delta}$, thereby ensuring $\mathcal{D} \models \overline{\exists}\mathit{ExplDiag}(\bar{\delta}, \phi, \bar{\eta})$. In the next section we will show how we can determine simpler parameter constraints Ψ for $\bar{\delta}$ than $\mathit{ExplDiag}(\bar{\delta}, \phi, \bar{\eta})$ along with the computation of $\bar{\delta}$.

When computing diagnoses and diagnosis templates we would like the computation to be guided by preference values. In general there may be instances of $\langle \bar{\delta}, \Psi\rangle$ that have different preference values. So, what is the preference value of $\langle \bar{\delta}, \Psi\rangle$? The minimum? The maximum? Our approach here is to consider only *preference-uniform* diagnosis templates, i. e., diagnosis templates $\langle \bar{\delta}, \Psi\rangle$ such that there exists a value v with $\wp_{\phi,\bar{\eta}}(\bar{\delta}') = v$ for all ground instances $\bar{\delta}'$ of $\langle \bar{\delta}, \Psi\rangle$. This value v is the preference value of the preference-uniform diagnosis template $\langle \bar{\delta}, \Psi\rangle$. Note that we do not miss any diagnosis when we restrict ourself to preference-uniform diagnosis templates: For every diagnosis $\bar{\delta}$ $\langle \bar{\delta}, \mathit{True}\rangle$ is a preference-uniform diagnosis template.

7 Appropriate Basic Action Theories

In this section we use a situation calculus function symbol pref denoting the preference valuation $\wp_{\phi,\bar{\eta}}$. Especially, $\wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ is denoted by $\mathit{pref}(\delta, \sigma)$ with $\sigma = \mathit{do}(\bar{\delta}', S_0)$. It is not necessary (but possible) to really formalize pref within $\mathcal{D}$ since all the places where we use pref in this paper are only intended to clarify our concepts.[9]

Given an observation ϕ and a history $\bar{\eta}$, our aim now is to compute preference-uniform explanatory history-based diagnosis templates $\langle \bar{\delta}, \Psi\rangle$ and their preference values. For this purpose we use action variation axioms of the form

$$\begin{aligned}\mathit{Varia}(a, A(x_1, \ldots, x_n), s) \equiv \quad & \Theta_A^{(1)}\lceil a, x_1, \ldots, x_n, s\rceil \\ & \vee \cdots \\ & \vee \Theta_A^{(k)}\lceil a, x_1, \ldots, x_n, s\rceil\end{aligned}$$

Each of the $\Theta_A^{(\ell)}\lceil a, x_1, \ldots, x_n, s\rceil$ has the form

$$\exists y_1, \ldots, y_m\, [a = \delta_A^{(\ell)} \wedge \theta_A^{(\ell)}\lceil y_1, \ldots, y_m, x_1, \ldots, x_n, s\rceil]$$

where $\delta_A^{(\ell)}$ is a (possibly non-ground) action, $y_1, \ldots, y_m$ are the variables occurring in $\delta_A^{(\ell)}$, and $\theta_A^{(\ell)}$ is a formula which is uniform in s. Note that

$$\mathcal{D} \models \overline{\forall}[\theta_A^{(\ell)} \supset \mathit{Varia}(\delta_A^{(\ell)}, A(x_1, \ldots, x_n), s)]$$

That is, $\theta_A^{(\ell)}$ states a condition under which $\delta_A^{(\ell)}$ is a valid variation of $A(x_1, \ldots, x_n)$ in situation s. The action variation axioms given in the situation calculus section are examples for that already, e. g.:

$$\begin{aligned}\mathit{Varia}(a, \mathit{putdown}(x), s) \equiv \quad & a = \mathit{putdown}' \\ & \vee \exists y\, [a = \mathit{putdown}(y) \wedge \mathit{Having}(y, s)]\end{aligned}$$

[9] After formalizing the range of $\wp_{\phi,\bar{\eta}}$, if t is a term for $\wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ and $\sigma = \mathit{do}(\bar{\delta}', S_0)$ we will have $\mathcal{D} \models \mathit{pref}(\delta, \sigma) = t$.

where $\delta^{(1)}_{putdown} = putdown'$, $\theta^{(1)}_{putdown} = True$,
$\delta^{(2)}_{putdown} = putdown(y)$, $\theta^{(2)}_{putdown} = Having(y,s)$.
In the next section these $\theta^{(\ell)}_A$ are used to build up the parameter constraints of diagnosis templates.

Furthermore we provide the algorithm with finite non-empty sets $X^{(\ell)}_A$ of pairs $\langle \chi\lceil y_1,\ldots,y_m,x_1,\ldots,x_n,s\rceil, w\rangle$ where $0 \le w \le 1$ and χ is a formula which is uniform in s. The intuition behind this is that χ is a condition under which the preference value is w, i. e.,

$$\mathcal{D} \models \overline{\forall}[\theta^{(\ell)}_A \wedge \chi \supset pref(\delta^{(\ell)}_A, s) = w]$$

Actually, the preference values $\wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ for variations δ are defined by the values given in the $X^{(\ell)}_A$. In the robot example (where $A(x) = putdown(x)$) we may have

$$X^{(1)}_{putdown} = \{\langle \neg Having(x,s), 1\rangle, \langle Having(x,s), w^{(1)}\rangle\}$$
$$X^{(2)}_{putdown} = \{\langle y = x, 1\rangle, \langle \neg Having(x,s) \wedge y \neq x, 1\rangle, \langle Having(x,s) \wedge y \neq x, w^{(2)}\rangle\}$$

$\langle y = x, 1\rangle$ is necessary to ensure the preference valuation to be inert. The other both 1's are chosen because the associated χ's are conditions for variations that are aftereffects: if the robot is not having x then $putdown(x)$ cannot be executed. Hence, for instance, when variations of $putdown(x)$ are concerned

$$\mathcal{D} \models \overline{\forall}[\neg Having(x,s) \supset pref(putdown', s) = 1]$$
$$\mathcal{D} \models \overline{\forall}[Having(y,s) \wedge Having(x,s) \wedge y \neq x \supset pref(putdown(y), s) = w^{(2)}]$$

To avoid contradiction it is necessary for $\langle \chi, w\rangle$ and $\langle \chi', w'\rangle$ in $X^{(\ell)}_A$ with $w \neq w'$ that χ and χ' exclude each other w. r. t. $\theta^{(\ell)}_A$, i. e., $\mathcal{D} \models \overline{\forall}[\theta^{(\ell)}_A \supset \neg(\chi \wedge \chi')]$. In order to provide a preference value for every diagnosis and diagnosis template we must require $\mathcal{D} \models \overline{\forall}[\theta^{(\ell)}_A \supset \bigvee_{\langle \chi, w\rangle \in X^{(\ell)}_A} \chi]$.

No wonder that we treat the insertion axiom similarly, i. e., we use an insertion axiom of the form

$$Inser(a,s) \equiv \Theta^{(1)}\lceil a,s\rceil \vee \cdots \vee \Theta^{(k)}\lceil a,s\rceil$$

Each of the $\Theta^{(\ell)}\lceil a,s\rceil$ has the form

$$\exists y_1,\ldots,y_m\,[a = \delta^{(\ell)} \wedge \theta^{(\ell)}\lceil y_1,\ldots,y_m,s\rceil]$$

where $\delta^{(\ell)}$ is a (possibly non-ground) action, $y_1,\ldots,y_m$ are the variables occurring in $\delta^{(\ell)}$, and $\theta^{(\ell)}$ is a formula which is uniform in s. Note that

$$\mathcal{D} \models \overline{\forall}[\theta^{(\ell)} \supset Inser(\delta^{(\ell)}, s)]$$

That is, $\theta^{(\ell)}$ states a condition under which $\delta^{(\ell)}$ is a valid insertion in situation s. The insertion axiom given in the situation calculus section is an example for that already:

$$\begin{aligned} Inser(a,s) \equiv \quad & \exists z\,[a = robotloses(z) \wedge Having(z,s)] \\ \vee\, & \exists z\,[a = somebodytakes(z) \wedge Portable(z)] \end{aligned}$$

where $\delta^{(1)} = robotloses(z)$, $\theta^{(1)} = Having(z, s)$,
$\delta^{(2)} = somebodytakes(z)$, $\theta^{(2)} = Portable(z)$.
In the next section these $\theta^{(\ell)}$ are used to build up the parameter constraints of diagnosis templates.

Furthermore we provide the algorithm with a finite non-empty set $X^{(\ell)}$ of pairs $\langle \chi\lceil y_1, \ldots, y_m, s\rceil, w\rangle$ where $0 \leq w \leq 1$ and χ is a formula which is uniform in s. The intuition behind this is that χ is a condition under which the preference value is w, i. e.,

$$\mathcal{D} \models \overline{\forall}[\theta^{(\ell)} \wedge \chi \supset pref(\delta^{(\ell)}, s) = w]$$

Actually, the preference values $\wp_{\phi,\bar{\eta}}(\delta \,|\, \bar{\delta}')$ for insertions δ are defined by the values given in the $X^{(\ell)}$. To avoid contradiction it is necessary for $\langle \chi, w\rangle$ and $\langle \chi', w'\rangle$ in $X^{(\ell)}$ with $w \neq w'$ that χ and χ' exclude each other w. r. t. $\theta^{(\ell)}$, i. e., $\mathcal{D} \models \overline{\forall}[\theta^{(\ell)} \supset \neg(\chi \wedge \chi')]$. In order to provide a preference value for every diagnosis and diagnosis template we must require $\mathcal{D} \models \overline{\forall}[\theta^{(\ell)} \supset \bigvee_{\langle \chi, w\rangle \in X^{(\ell)}} \chi]$.

Basic action theories whose action variation axioms and insertion axiom are in such a way are appropriate for computing preference-uniform diagnosis templates as is shown in the next section. Note that with this approach valid variations and insertions are independent of the observation. It is a topic under investigation how the given observation can be exploited to refine the computation of diagnosis templates.

8 Computing Diagnosis Templates

With appropriate basic action theories at hand we are able to define a search space in order to compute preference-uniform explanatory history-based diagnosis templates $\langle \bar{\delta}, \Psi\rangle$ and their preference values for given observation ϕ and history $\bar{\eta} = [\eta_1, \ldots, \eta_n]$. Each state in the search space is a quadruple $\langle \bar{\delta}, \Psi, v, j\rangle$ where

- $\bar{\delta}$ is a first part of an extended variation of the history
- Ψ is a parameter constraint for $\bar{\delta}$
- v is the preference value assigned with $\langle \bar{\delta}, \Psi\rangle$
- j is the number of the last history action that is "covered" by $\bar{\delta}$
  (i. e.: In $\bar{\delta}$ there are variations of the first j history actions.)

A successor of a state $\langle \bar{\delta}, \Psi, v, j\rangle$ is a state $\langle \bar{\delta}', \Psi', v', j'\rangle$ such that (with the notations as above)

- $\bar{\delta}' = \bar{\delta}{\scriptstyle\bullet}\delta_A^{(\ell)\prime}$
  $\Psi' = \Psi \wedge \theta_A^{(\ell)}(y_1', \ldots, y_m', t_1, \ldots, t_n, do(\bar{\delta}, S_0))$
  $\quad\quad \wedge \chi(y_1', \ldots, y_m', t_1, \ldots, t_n, do(\bar{\delta}, S_0))$
  $v' = v \cdot w$
  $j' = j + 1$
  where $\langle \chi, w\rangle \in X_A^{(\ell)}$,
  $\quad y_1', \ldots, y_m'$ are new variables replacing the variables of $\delta_A^{(\ell)}$ yielding $\delta_A^{(\ell)\prime}$
  and $\eta_{j+1} = A(t_1, \ldots, t_n), j < n$

or

– $\bar{\delta}' = \bar{\delta}.\delta^{(\ell)\prime}$
$\Psi' = \Psi \wedge \theta^{(\ell)}(y'_1, \ldots, y'_m, do(\bar{\delta}, S_0))$
$\quad\quad \wedge \chi(y'_1, \ldots, y'_m, do(\bar{\delta}, S_0))$
$v' = v \cdot w$
$j' = j$
where $\langle \chi, w \rangle \in X^{(\ell)}$,
$\quad y'_1, \ldots, y'_m$ are new variables replacing the variables of $\delta^{(\ell)}$ yielding $\delta^{(\ell)\prime}$

Actually, instead of directly assigning $\Psi' = \Psi \wedge \theta \wedge \chi$ the assignment is $\Psi' = \Psi \wedge \theta$ if $\mathcal{D} \models \overline{\forall}[\Psi \wedge \theta \supset \chi]$ what is checked first. The initial state is $\langle [\,], True, 1, 0 \rangle$. A state $\langle \bar{\delta}, \Psi, v, j \rangle$ is defined to be a goal state iff $\mathcal{D} \models \overline{\exists}[\Psi \wedge Exec(\bar{\delta}) \wedge \phi[\bar{\delta}]]$ and $j = n$.[10]

Lemma 15. If $\langle \bar{\delta}, \Psi, v, j \rangle$ is a goal state then $\langle \bar{\delta}, \Psi \rangle$ is a preference-uniform explanatory history-based diagnosis template for ϕ and $\bar{\eta}$ with preference value v. ∎

Theorem 16. Each diagnosis with preference value v is an instance of some diagnosis template $\langle \bar{\delta}, \Psi \rangle$ such that $\langle \bar{\delta}, \Psi, v, n \rangle$ is a goal state. ∎

There are possibilities to prune the search space: If $\langle \bar{\delta}, \Psi, v, j \rangle$ is a goal state with $\bar{\delta} = [\delta_1, \ldots, \delta_m]$ then $\Psi = True \wedge \Psi_1 \wedge \cdots \wedge \Psi_m$ where each Ψ_i corresponds to δ_i. Since $Exec(\bar{\delta}) = Poss(\delta_1, \sigma_1) \wedge \cdots \wedge Poss(\delta_m, \sigma_m)$ with $\sigma_i = do([\delta_1, \ldots, \delta_{i-1}], S_0)$ and $\mathcal{D} \models \overline{\exists}[\Psi \wedge Exec(\bar{\delta})]$ must hold for a goal state $\mathcal{D} \models \overline{\exists}[\Psi_i \wedge Poss(\delta_i, \sigma_i)]$ are necessary conditions for being a goal state as well as $\mathcal{D} \models \overline{\exists}[\Psi_1 \wedge \cdots \wedge \Psi_i \wedge Poss(\delta_i, \sigma_i)]$. These conditions can be tested when generating the successors of a search state. This analysis also yields that $\mathcal{D} \models \overline{\exists}[\Psi' \wedge Exec(\bar{\delta}')]$ is a (stronger) necessary condition when generating the successor $\langle \bar{\delta}', \Psi', v', j' \rangle$ of $\langle \bar{\delta}, \Psi, v, j \rangle$.

The search space given here is well suited for search algorithms that work like uniform-cost search except that here the nodes in the search tree that have higher preference values have to be expanded first. Note that the branching factor is finite since there are only finitely many $\delta_A^{(\ell)}$ and $\delta^{(\ell)}$ and all of the $X_A^{(\ell)}$ and $X^{(\ell)}$ are finite. We have implemented a prototypical diagnostic system in PROLOG that uses an algorithm with iterative deepening along the preference values. It also computes for each diagnosis template all ground instances that are diagnoses.

The algorithm is sound and optimal (in the sense that it first outputs diagnosis templates with highest preference value). There are weak conditions under which the algorithm is complete: Assumed that a diagnosis template with non-zero preference value exists, it is sufficient for completeness that $w < 1$ for each $\langle \chi, w \rangle$ in each $X^{(\ell)}$. This restriction is not necessary for the $X_A^{(\ell)}$.

We ran several experiments[11] in order to get an estimate how computing diagnosis templates compares with computing diagnoses. For the latter we used an algorithm similar to the one outlined here and in [7].[12] In these experiments runtimes were measured for computing

[10] If $j = n$ then $\mathcal{D} \models \overline{\forall}[\Psi \supset (ExtVari(\bar{\delta}, \bar{\eta}) \wedge pref(\bar{\delta}) = v)]$ because of the construction.

[11] on a Pentium 500 Mhz Linux-PC using ECLiPSe 4.2

[12] Instead of actions δ and constraints $\theta \wedge \chi$, ground instances of $\langle \delta, \theta \wedge \chi \rangle$ are used and constraints omitted when computing successors of search states.

- all diagnoses with maximal preference value
- all diagnosis templates with maximal preference value
  and all their ground instances that are diagnoses

and computing diagnoses directly took 3.65 to 8.33 times more runtime than computing them via diagnosis templates. For short histories (like in the robot example so far) the system outputs all most preferred diagnoses almost instantaneously (within less than 0.1 seconds). In another robot example the history consists of 42 actions and 2.36 minutes were needed (compared to 14.5 minutes without using templates). In both robot examples the system has to deal with 10 rooms and 1000 books.

Of course, these experiments only provide a first glance on potential runtime savings. We are working on a formal analysis of the complexity and on testing our approach with some "benchmark problems" and comparing it with other approaches in order to strengthen the experimental results.

9 Summary and Future Work

Our concern in this paper has been to formalize how a certain observation can be explained by answering the question of what happened instead of an assumed history if this assumed history contradicts the observation. We introduced the notion of history-based diagnosis in the situation calculus and demonstrated the benefit of this approach for diagnosis in dynamic domains. History-based diagnoses are possible histories that are extended variations of the assumed history. Furthermore we showed how diagnosis preference criteria can be described using preference values and how preference values of diagnoses can be calculated from preference values for single modifications of the history.

In order to compactly represent sets of history-based diagnoses that are all instances of one action sequence history-based diagnosis templates can be used which consists of that action sequence and a formula constraining the possible instances of the action sequence. It is shown that, with an appropriate basic action theory at hand, history-based diagnosis templates can be computed taking advantage of the structure of the basic action theory's axioms. We outlined an algorithm guaranteeing to find the most preferred diagnosis templates.

This is work in progress and a lot remains to be done. We already mentioned that we intend to refine the computation of diagnosis templates by paying more attention to the given observation. The topics currently under investigation also include, e. g., multiple observations (at several points in the history), testing and repair (incl. sensing). In a different framework, these topics are addressed in [1] in order to characterize diagnostic problem solving whereas we also have in mind computational aspects. Another difference is that their approach does not allow to consider action variations which we regard as an important feature of our approach.

References

1. C. Baral, S. McIlraith, and T.C. Son. Formulating diagnostic problem solving using an action language with narratives and sensing. In *Proc. 7th Intern. Conf. on Principles of Knowledge Representation and Reasoning*, 2000.
2. C. Bastié and P. Régnier. SPEEDY: Monitoring the execution in dynamic environments. In *Proc. Workshop on Reasoning about Actions and Planning in Complex Environments*, 1996.
3. W. Burgard, A.B Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1–2), 1999.
4. M.-O. Cordier and S. Thiébaux. Event-based diagnosis for evolutive systems. In *Work. Pap. 5th Intern. Workshop on Principles of Diagnosis*, 1994.
5. G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In *Proc. 6th Intern. Conf. on Principles of Knowledge Representation and Reasoning*, 1998.
6. K.Z. Haigh and M.M. Veloso. Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, 5(1), March 1998.
7. G. Iwan. Explaining what went wrong in dynamic domains. In *Proc. 2nd Intern. Cognitive Robotics Workshop*, 2000.
8. H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(018), 1998.
9. J. McCarthy. Situations, actions and causal laws. Stanford Artificial Intelligence Project: Memo 2, 1963. Reprinted in [12].
10. S.A. McIlraith. Towards a theory of diagnosis, testing and repair. In *Work. Pap. 5th Intern. Workshop on Principles of Diagnosis*, 1994.
11. S.A. McIlraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *Proc. 6th Intern. Conf. on Principles of Knowledge Representation and Reasoning*, 1998.
12. M.L. Minsky, editor. *Semantic Information Processing*. MIT Press, Cambridge, MA, 1968.
13. R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. Academic Press, San Diego, CA, 1991.

A Defense Model for Games with Incomplete Information

Wojciech Jamroga

Parlevink Group, University of Twente, Netherlands
jamroga@cs.utwente.nl
http://www.cs.utwente.nl/~jamroga

Abstract. Making a decision, an agent must consider how his outcome can be influenced by possible actions of other agents. A 'best defense model' for games involving uncertainty assumes usually that the opponents know everything about the actual situation and the player's plans for certain. In this paper it's argued that the assumption results in algorithms that are too cautious to be good in many game settings. Instead, a 'reasonably good defense' model is proposed: the player should look for a best strategy against all the potential actions of the opponents, still assuming that any opponent plays his best *according to his actual knowledge*. The defense model is formalized for the case of two-player zero-sum (adversary) games. Also, algorithms for decision-making against 'reasonably good defense' are proposed.
The argument and the ideas are supported by the results of experiments with random zero-sum two-player games on binary trees.

1 Introduction

Under uncertainty, an agent must consider all the possible situations, called often the *possible worlds*. Although he might not be able to distinguish between many of them, the actual 'state of affairs' severely influences the future course of action and the final income the agent is going to gain.

A two-player poker game may be a good example. The set of possible worlds Ω simply consists of all the possible card distributions. Suppose that MAX[1] has ♡AKJ8 ♣7 in the actual game. Then MAX can restrict his reasoning to the situations that seem plausible to him – namely, all the distributions where MAX has ♡AKJ8 ♣7 (see figure 1). In most situations MAX can't identify MIN's actual beliefs, because he doesn't know which cards MIN actually possess. Note however, that *if* MAX knew the actual world precisely, he would try to figure out the set of situations that seem plausible to MIN, as well as MIN's future line of action. The set would consist of all the distributions where MIN has a

[1] The agent of concern is often called the *MAX* player (since he should maximize his output) in the theory of zero-sum games. His opponent is labeled *MIN* then – he maximizes *his* output, which means that he tries to minimize the resulting score of MAX.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 260–274, 2001.

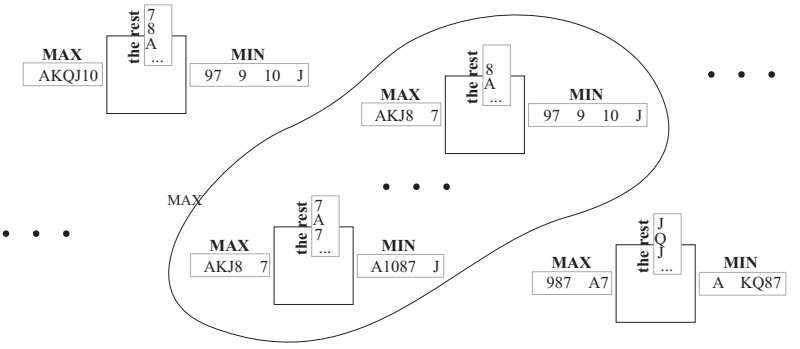


Fig. 1. The set of possible worlds Ω. In the actual game MAX has ♡AKJ8 ♣7, so he can restrict the set to Ω_{MAX}.

particular hand of five cards (in the example on figure 2: ♠97 ♡9 ♢10 ♣J). Note that such a hand must contain no cards possessed by MAX because MAX *knows* that MIN can't have them.

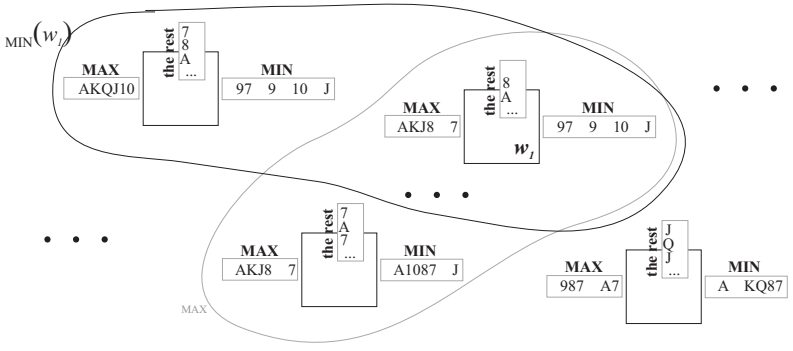


Fig. 2. The set of plausible worlds according to MIN – if w_1 is the actual distribution.

This is the idea underlying the decision-making algorithms *gvm* and *find-optimal*, proposed in this paper. The player can consider every world from Ω separately – identifying the possible distributions of resources as well as possible beliefs of the opponents. Then he can choose the action that gives him the highest expected outcome over all the worlds.

2 Best Defense

In Game Theory, a player is assumed to play against optimal defense, since a 'rational opponent' makes always the best decision. For zero-sum games this implies that the opponent chooses the worst move from the player's perspective. A best defense model for imperfect information games was proposed in (Frank 1996),

(Frank & Basin 1998a) and (Frank & Basin 1998b). It contains the following assumptions:

1. MIN has perfect information about the situation,
2. MIN knows MAX's actual strategy,
3. MAX's knowledge is limited to just knowing the set of all possible situations (worlds) Ω,
4. the strategy adopted by MAX must be a pure strategy.

MAX maximizes his expected payoff value (over Ω). The opponent (MIN) is assumed to be omniscient; thus, he can maximize his payoff directly.

Since the problem of finding optimal strategy (even in such a simplified setting) is NP-complete, there is a strong need for suboptimal but less complex algorithms. A number of minimaxing algorithms – including vector minimaxing (*vm*) and payoff-reduction minimaxing (*prm*) – were proposed in (Frank, Basin & Matsubara 1998) and (Frank & Basin 1998b). The algorithms were then compared to the algorithm of Monte Carlo sampling (*MC*),[2] based on classical minimaxing. In the competition an algorithm was claimed better if it was finding strategies close to optimal more frequently than its competitor – within the notion of 'optimality' defined above. In a series of experiments on random tree games, Monte Carlo sampling algorithm was definitely outperformed by *prm* (and it turned out to be slightly worse than *vm*, too). However, it's not clear why an algorithm that plays very well against an omniscient opponent should also win in a more realistic competition.

2.1 Experiments with Random Games

The experiment idea is strongly based on the experiments done by (Frank, Basin & Matsubara 1998). The test was conducted for games on complete binary trees of depth D. For any of N possible worlds from Ω a payoff is assigned to every tree leaf; the payoff may be either 0 or 1. If a game ends in leaf l and world w appears to be the actual world, the player wins the payoff value assigned to (l, w), and the opponent loses the same amount. A possible world is described with a list of payoffs for all the tree leaves in this world, called a *payoff vector*. Thus, to generate a random game, one must generate a random payoff vector for each world. Note that two different worlds can have same payoff vectors.

An example of such a game is shown on figure 3.

To make this a fair competition between *algorithms* – the competitors must be provided with equal chances of winning. Say the algorithms are called A and B. Now, for a particular (randomly generated) game:

1. first A plays as MAX and B as MIN. The strategies are identified and the expected value of payoff computed (over Ω);
2. then *the same* game is played again – A plays as MIN and B as MAX;
3. at the end the difference between payoffs is computed. If the difference is positive, A wins; if it's negative then B is the winner.

[2] (Corlett & Todd 1985), (Ginsberg 1999)

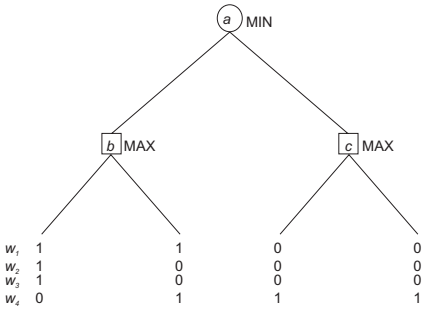


Fig. 3. A game tree of depth $D = 2$ with $N = 4$ possible worlds

1000 games with MAX as a starting player and 1000 games with MIN as a starting player were played during every such competition. The algorithms involved in the competition were: *MC*, *vm*, *prm*, and the simplest algorithm for finding the optimal strategy against 'best defense' – checking all the possible player's strategies one by one (let's name it *opt-bd*, for instance). The output of every competition is described by A's 'triumph supremacy' (number of rounds won by A minus rounds lost by A) and A's payoff supremacy (the average expected payoff value per 1000 rounds). Most experiments were conducted for games with $D = 8, N = 1000$, except the competitions involving *opt-bd* algorithm – $D = 4, N = 1000$ (analyzing any game of more than 4 turns is practically infeasible for *opt-bd*). The results of the actual experiments are shown on figure 4.

	triumph supr.	payoff supr.
MC vs. vm	3.4%	0.2
MC vs. prm	39.1%	9.2
vm vs. prm	33.9%	8.0

	triumph supr.	payoff supr.
MC vs. opt-bd	36.3%	7.1
vm vs. opt-bd	38.4%	8.1
prm vs. opt-bd	19.1%	4.2

Fig. 4. The competition output: triumph supremacy (in [%] of total rounds played) and payoff supremacy (per 1000 rounds). Experiment setting: $N = 1000$ worlds; tree depth $D = 8$ (array on the left), $D = 4$ (array on the right).

However, the setting of the experiment above may be somewhat misleading since we implicitly assumed that both agents have the same knowledge. In real situations most agents can at least eliminate some of the worlds from Ω as being impossible. For instance, in card games every agent holds some cards in his hand. He knows the cards he has, so he can exclude all the card distributions inconsistent with this knowledge. Since different players have access to different pieces of the reality – the worlds actually possible for MAX ($\Omega_{MAX} \subset \Omega$) and for MIN ($\Omega_{MIN} \subset \Omega$) should differ in most cases.

New experiment: MAX and MIN find their strategies with respect to separate sets of worlds considered to be possible. Ω_{MAX} and Ω_{MIN} are generated on

random for every game. The players' knowledge is assumed to be adequate, i.e. the output is computed as the expected value of payoff over the worlds from $\Omega_{MAX} \cap \Omega_{MIN}$ only. Since there is one more random factor in the setting, 20000 games per round are played instead of 2000. The results are shown on figure 5.

	triumph supr.	payoff supr.
MC vs. vm	2.1%	2.1
MC vs. prm	21.7%	17.7
vm vs. prm	21.7%	14.1

	triumph supr.	payoff supr.
MC vs. opt-bd	18.8%	12.1
vm vs. opt-bd	20.4%	13.0
prm vs. opt-bd	10.6%	7.0

Fig. 5. The competition again: players' belief sets are generated randomly. Experiment setting: $N = 1000$ worlds; tree depth $D = 8$ (array on the left), $D = 4$ (array on the right).

The results of the experiments show that it doesn't have to be beneficial for a player to assume that the opponent reaches the upper bound of his theoretical capabilities (especially in the context of his knowledge about the actual situation). The cautious algorithms: *prm* and *opt-bd* were in fact outperformed by Monte Carlo sampling, which was considered very suboptimal. A possible reason lies in incoherence of the adopted best defense assumptions with the situations being encountered in the actual games.

In perfect information games the opponent (given sufficient resources) plays sub-optimally only by his own fault. He can always use the best defense strategy since he can find it by minimaxing. The agents can always play best defense in perfect information games (just by finding the strategies with minimax). On the other hand, in games with incomplete information the opponent is seldom able to fulfill the 'best defense' assumption because his knowledge is insufficient. Thus, the model makes the player assume a defense which is impossible to be met in most cases.

3 Reasonably Good Defense

As the experiments showed, it is not beneficial for the player to overestimate capacities of the opponent too much. The best defense model by Frank & Basin refers clearly to the worst possible line of events, but this line is quite unlikely to occur.

In a probabilistic framework a model of MIN's beliefs is necessary. The model should include MIN's beliefs about the actual situation as well as beliefs about the player's beliefs. The beliefs may depend on the actual world and the state of the game. MIN maximizes his expected payoff over Ω with respect to his actual state of belief (i.e. he minimizes the payoff for MAX in zero-sum games). MAX should maximize his expected payoff over Ω and the set of possible MIN belief states.

Reasonably good defense model:

> If nothing suggests the contrary, the opponent should be assumed capabilities similar to the player. Thus, MAX's knowledge and skills, and his model of MIN should be symmetrical.

In particular – if (in a given situation) no specific knowledge is available about likelihood of some possible worlds or different opponent beliefs, equal probabilities should be supposed a priori, also when modeling the beliefs of other agents.

3.1 A Simple Case

The problem is often analyzed in a simplified version, when all the worlds are equally probable by rule, but the agents can identify some of them as implausible at some point of a game. MAX's knowledge can be described as $\Omega_{MAX} \subset \Omega$. The player can't know which worlds $w \in \Omega$ are considered to be plausible by the opponent. However, he can restrict the set of possible MIN's belief sets Ω_{MIN} to those beliefs which are consistent with his own knowledge – like in the example presented on figures 1 and 2 back in section 1.

To evaluate a MIN node s, MAX may consider evaluations for all the possible opponent's belief sets Ω_{MIN}. (Gambäck et al. 1993) present an interesting example of such an analysis, concerning Bridge bidding. The player sees his own cards (hand) – so he can generate possible distributions from Ω_{MAX} by assigning the remaining cards to the other players (the authors call each of the alternative assignments an *R-deal*). Every world (R-deal) $w \in \Omega_{MAX}$ determines a MIN's belief set $\Omega_{MIN}(w)$ – namely, $\Omega_{MIN}(w)$ is a set of worlds which cannot be distinguished from w by the opponent (in the actual state of the game). In the case of Bridge bidding, for instance $\Omega_{MIN}(w)$ consists of all the distributions w' in which MIN has exactly the same cards as in w. Whenever MAX needs to consider opponent's decisions he can model the opponent's view by generating $\Omega_{MIN}(w)$ for each $w \in \Omega^0_{MAX}$ (since analogous function $\Omega_{MAX}(w)$ is needed to model the opponent's knowledge about the player's possible beliefs, let's rather call the MAX's actual belief Ω^0_{MAX} to avoid confusion).

Note that the actual shape of $\Omega_{MIN}(w)$ depends on the game rules. For instance, two players can't possess the same card in a poker game. So if MIN has ♢A1087 ♣J in a world w then $\Omega_{MIN}(w)$ includes all the situations of MIN having exactly ♢A1087 ♣J, and MAX having none of the cards (the rest of the deck must contain none of these cards, too). However, this would not work for Canasta, where two complete decks of cards are mixed and dealt – so two different players can even possess hands of the same shape!

An algorithm for finding the decision against 'reasonably good defense' (2 players, zero-sum game, no information about worlds' likelihood except that some worlds are actually impossible; the player's belief doesn't change when moving to another game state – no information flow) is shown on figure 6. *Generalized vector minimaxing* (*gvm*) is a more universal version of algorithms like Monte Carlo or vector minimaxing from (Frank et al. 1988). The algorithm has been inspired by ideas from (Carmel & Markovitch 1996) and (Gambäck et al. 1993) – the player looks forward for his opponent's decision in every possible situation, and then maximizes his expected output against such defense. *Gvm* allows to model the players' knowledge on any arbitrary level, since the functions Ω_{MIN}, Ω_{MAX} are assumed to mutually encode a player's beliefs about his

gvm (Game, s, player, Worlds);

Generalized vector minimaxing. Returns the evaluation vector $(eval[w_1], ..., eval[w_n])$ for node s, together with the player's chosen move. Parameters:

Game: game definition, including functions $Succ(s)$ – returning the set of successors for node s, $Strat(player)$ – returning the set of all the possible player's (complete) strategies, $payoff(l)$ – returning the MAX's payoff vector $(payoff(l)[w_1], ..., payoff(l)[w_n])$ in leaf l, $\Omega_{MIN}(w)$, $\Omega_{MAX}(w)$ – returning MIN's and MAX's beliefs in a particular world w;

s: game state (node);

player: the agent who makes a decision at this node (MAX or MIN);

$Worlds$: the agent's actual belief (Ω^0_{MAX} or Ω^0_{MIN} in this case);

if $Succ(s) = \emptyset$ then return $(nil, payoff(s))$;
else:

- for every $s' \in Succ(s)$ compute $e_{s'} = (e_{s'}[w_1], ..., e_{s'}[w_n])$ as:
  $$e_{s'}[w] = \begin{cases} gvm(Game, s', \text{MIN}, \Omega_{MIN}(w))[w] & \text{if } player = \text{MAX} \\ gvm(Game, s', \text{MAX}, \Omega_{MAX}(w))[w] & \text{if } player = \text{MIN} \end{cases}$$
  for every world $w \in \Omega$;
- if *player*=MIN then return $(s', e_{s'})$ such that $\sum_{w \in Worlds} e_{s'}[w]$ is minimal. else return $(s', e_{s'})$ such that $\sum_{w \in Worlds} e_{s'}[w]$ is maximal.

Fig. 6. Generalized vector minimaxing.

opponent's beliefs as well as his beliefs about his opponent's beliefs about his beliefs etc.

Note that if

- $\Omega_{MAX}(w) = \Omega^0_{MAX}$ (the opponent knows the player's state of belief)

then $gvm(Game, s, MAX, \Omega^0_{MAX})$ returns the same strategy as the *vector minimaxing* algorithm proposed by Frank, Basin & Matsubara. If we also assume that

- $\Omega_{MIN}(w) = \{w\}$ (the opponent always knows the actual situation),

we obtain the instance of vector minimaxing that was actually used in (Frank et al 1998).

On the other hand, if the game definition includes the following assumptions:

- $\Omega_{MIN}(w) = \{w\}$,
- $\Omega_{MAX}(w) = \{w\}$;

then *gvm* becomes equivalent to classical Monte Carlo minimaxing.

The main disadvantage of *gvm* is that it's not always able to find the optimal strategy – due to *non-locality*, a phenomenon observed originally in (Frank 1996), and formalized in (Frank & Basin 1998a). The game tree on figure 3 demonstrates the phenomenon well. Consider MAX's decision at node b. If the analysis was to be 'local', MAX would have to prefer the left-hand branch, since his expected

payoff equals 0.75 then (against 0.5 when following the right-hand branch). But making decision at node b means that MIN has made his decision to move to b, not to c, at node a. If we assume that MIN is rational, then his decision must make sense – and it makes sense only when he believes that worlds w_1, w_2, w_3 are irrelevant. In other words, w_4 is obviously the only world considered possible by MIN at this moment. If we assume that his beliefs are adequate (they can be incomplete, but never false), then MAX has to restrict his computation to w_4 alone, and therefore to pick up the right branch.

The implication of non-locality is that any 'compositional' algorithm that looks only forward, not backward, is bound to be suboptimal. Thus, the player should evaluate his decisions against whole strategies of the opponent, not their parts only. On the other hand, it's not possible to simulate the opponent's minimaxing over the whole game tree, because this would lead to an infinite loop.

Figure 7 presents an algorithm for finding the optimal strategy against reasonably good defense. The algorithm is based on the equilibrium definition for zero-sum games. It computes the minmax and the maxmin over the sets of players' strategies, and if they lead to the same result then the optimum has been found. Unfortunately, there is often no such an optimum. In this case *findoptimal* returns the minmax strategy, which describes the lower bound of the outcome the player can expect (since the assumption that the opponent always knows the player's strategy beforehand defines the upper bound of the opponent's knowledge). Another drawback of the algorithm is its computational complexity.

Note also that if the game definition includes the following assumptions:

1. $\Omega_{MIN}(w) = \{w\}$,
2. $\Omega_{MAX}(w) = \Omega^0_{MAX}$;

then $findoptimal(Game, MAX, \Omega^0_{MAX})$ computes the optimal strategy with respect to the 'best defense model' by Frank and Basin. In this sense their 'best defense' is a special case of 'reasonably good defense'. Yet the opponent's omniscience is not assumed (in practice) to be an inherent property of games, but has to be stated explicitly via Ω_{MIN}, Ω_{MAX} functions definition. Moreover, the algorithm indicates whether it's necessary to make any stronger claims about the opponent's knowledge to obtain a solution.

Finally, it is worth noting that – while *gvm*, as a minimaxing algorithm, has to be suboptimal for games with incomplete information – it can probably be improved in terms of accuracy. It demands for some reduction of the impact of non-locality on the decision-making process. Frank, Basin and Matsubara has already done it for traditional minimaxing within their 'best defense model' – *prm* algorithm is one of the results.

3.2 Computational Complexity

Not surprisingly, *opt-bd* is highly inefficient since it checks every possible player's strategy – its complexity is doubly exponential on the tree depth and linear on the number of worlds (namely, $o(N * b^{b^D})$ – where b stands for the branching

findoptimal (Game, player, Worlds);

The function returns the player's chosen strategy. It indicates also whether the strategy is optimal, or if it refers to the lower bound of the player's actual output (when the optimum doesn't exist). The algorithm presented below defines the function for $player = MAX$. If $player = MIN$, the algorithm is quite analogous.

[Function *output* computes the payoff value given a complete strategy pair and a particular world from Ω.]

For every $str \in Strat(MAX)$:

- for every $w \in \Omega$: let $strvect[w] = str$;
- for every $w \in Worlds$: $min[w] = maximize(MIN, strvect, \Omega_{MIN}(w))$;
- let $eval[str] = \sum_{w \in Worlds} output(str, min[w], w)$;

Let str_1 be that str for which $eval[str]$ is maximal;

For every $strvect \in \{\Omega \rightarrow Strat(MIN)\}$ such that $\forall_{w,v}\ (v \in \Omega_{MIN}(w) \Rightarrow strvect(v) = strvect(w))$:

- for every $v \in Worlds$: $max[v] = maximize(MAX, strvect, \Omega_{MAX}(v))$;
- let $eval[strvect] = \sum_{w \in Worlds} \sum_{v \in \Omega_{MIN}(w)} output(max[v], strvect[v], v)$;

Let $strvect_2$ be that $strvect$ for which $eval[strvect]$ is minimal;

if $eval[str_1] = max_{str}\{\sum_{w \in Worlds} output(str, strvect_2[w], w)\}$

- then return (str_1, OPTIMUM);
- else return (str_1, BOUND);

maximize (Game, player, strvect, Worlds);

The function searches the set of all *player*'s strategies, trying to maximize the expected payoff value over the given set of possible worlds against given opponent's strategy vector. Of course, MIN player wants to maximize *his* own payoff, i.e. to minimize the score defined by the *payoff* function.

Argument *strvect* is a function of type $\Omega \rightarrow Strat(MIN)$ for $player = MAX$, and $\Omega \rightarrow Strat(MAX)$ for $player = MIN$

if $player = MAX$ then:

- return the strategy $str \in Strat(MAX)$ for which $\sum_{w \in Worlds} output(str, strvect[w], w)$ is maximal;
- else: return the strategy $str \in Strat(MIN)$ for which $\sum_{w \in Worlds} output(strvect[w], str, w)$ is minimal;

Fig. 7. Algorithm for finding the optimal play against *reasonably good defense*.

factor of the game tree).[3] That's why suboptimal algorithms are useful: *MC* and *vm* as well as *prm* have the time complexity of $o(N * b^D)$ (exponential on the tree depth, linear on the number of worlds).

When the opponent's beliefs are taken into account, the complexity increases. The complexity of *findoptimal* algorithm is $o(N^2(b^{\frac{b^D}{2}})^{N+1})$, so in practical applications a suboptimal (but faster) algorithm is necessary. The complexity of *gvm* is exponential on the tree depth and polynomial on the number of worlds: $o(N^D * b^D)$. The practical complexity of the algorithm may be slightly reduced if we assume that players' beliefs must be adequate: $w \in \Omega_{MIN}(w)$, $w \in \Omega_{MAX}(w)$ for every w. Then the construction of evaluation vectors can be restricted to $w \in Worlds$ only (instead of all $w \in \Omega$).

There is a number of methods that can be used to reduce the search time at the expense of its accuracy. Sampling is often used to make the search feasible when the number of possible situations is huge; Gambäck, Rayner and Pell (Gambäck et al. 1993) propose such an approach for the case of bridge bidding, and Ginsberg's successful GIB program (Ginsberg 1999) uses Monte Carlo sampling in the complex domain of bridge card play. Evaluation approximator may help to keep the search depth at a reasonable level – Gambäck et al. used a trained neural network to implement such an approximation function, and they reported good results. Also, pruning techniques and heuristic search can be used for most domains of application.

3.3 More Experiments...

To test the new ideas against existing minimaxing algorithms, random binary tree games can be used again. To provide a natural interpretation to the belief functions Ω_{MIN}, Ω_{MAX} every game is treated as an 'imaginary card game'. Every world from Ω is defined by two hands of c 'cards' – one hand for each player. The deck consists of n 'cards'. A player can play only either the lowest or the highest card he possesses at the moment (when it's his turn to move, of course), so at any node (except the leaves) there are exactly two alternative decisions that can be made, regardless of the actual hand the player possesses. The payoffs for every leaf and each possible world are generated at random.

	triumph supr.	payoff supr.
gvm vs. MC	99.8%	94.9
gvm vs. prm	98.1%	76.0
gvm vs. vm	99.8%	90.5
gvm vs. opt-bd	98.9%	83.8

	triumph supr.	payoff supr.
MC vs. vm	2.7%	1.1
MC vs. prm	13.6%	5.2
vm vs. prm	34.8%	12.9
MC vs. opt-bd	77.2%	38.3
vm vs. opt-bd	85.9%	46.6
prm vs. opt-bd	80.8%	30.6

Fig. 8. Example results for $n = 7$, $c = 3$ cards (tree depth $D = 4$, $N = 140$ possible worlds).

[3] in the case of the experiments here: $b = 2$.

Now, $\Omega_{MIN}(w)$ is the set of worlds that cannot be excluded by MIN player in world w. Namely, it consists of these worlds that assume MIN having the hand he actually has. MIN's beliefs about MAX's beliefs, $\Omega_{MAX}(w)$ are defined in the same way. It is assumed that the players lead their cards secretly, i.e. the opponent doesn't know what card was played exactly – he knows only whether it was the highest or the lowest one (the actual world is recognized by both players no sooner than at the end of the game). Game parameters are: the tree depth $D = 2(c-1)$, and the number of possible worlds $N = \binom{n}{c} \cdot \binom{n-c}{c}$.

The *gvm* algorithm is played against other algorithms in a way similar to the experiments before. For a particular game, the game is played for every world (card distribution), and then the average value of payoff is computed. 1000 games are played for every competition: 500 with MAX as the leading player, and 500 with MIN starting the game (only for $n = 8, c = 3$, 200 games has been played due to complexity reasons). The results (triumph supremacy in [%] of total rounds played; payoff supremacy – average/estimated payoff per 1000 rounds) are shown on figures 8 and 9. Figure 8 shows also some example results of a competition between the traditional algorithms to make the comparison more thorough.

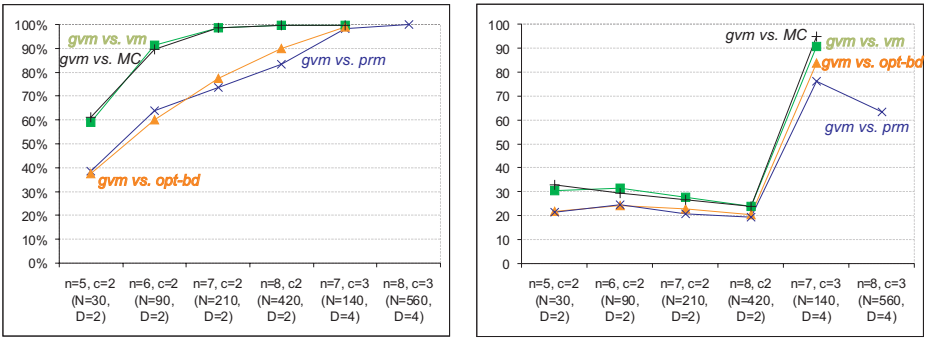


Fig. 9. Triumph supremacy in [%] of total rounds played (left), and Payoff supremacy per 1000 rounds (right).

In every competition *gvm* appeared to be at least 37.5% better than any of the other algorithms (in terms of the triumph supremacy). Moreover, as the game complexity increases, *gvm* starts to win practically 100% rounds.

The results reveal that algorithms like *prm* or *opt-bd* loose less than *MC* or *vm* when played against *gvm*. However, when played against each other, the previous pattern still holds: *MC* wins with *vm*, *prm* and *opt-bd*, *vm* wins with *prm* and *opt-bd* etc. The reason lies probably in the fact that *prm* and *opt-bd* were designed to play against a considerably more potent opponent. Thus, playing against *gvm* they can benefit from their cautiousness. On the other hand, *MC* and *vm* are apparently better off in games against an enemy of the same or similar level of skill and knowledge.

Another observation may be more surprising. When the tree depth increases, the gap between the results of *gvm* grows rapidly – in terms of the payoff supremacy as well as the frequency of winning. However, for a fixed D, *gvm* seems to earn less average payoff if the number of possible worlds increases, while still winning more and more rounds. This means that *gvm* succeeds to find a superior strategy for more initial 'hands of cards', but its expected payoffs for every particular hand decrease. The reason is perhaps that when N grows, more worlds are possible for any particular hand. Even a good algorithm can't play for all the worlds at the same time, so it is bound to fail in quite a number of them. Thus, an average difference in payoffs decrease, although *gvm* is still able to find a strategy better than the others. Moreover, when the tree depth is small, there is a very limited amount of different payoff vectors available. Now, when the number of worlds being considered at every node increase, it becomes more likely that the actual $Worlds$ set may be similar to some of $\Omega_{MIN}(w)$ and/or $\Omega_{MAX}(w)$ sets. Which means that *gvm* minimaxes over similar payoff vectors as its competitors in many games.

3.4 Generalizations

The games analyzed so far were constrained by several important simplifications. More realistic setting should include the following issues:

- for a game node (state) s: not every move (arc) can be taken in a particular situation $w \in \Omega$. Example: a player can lead A♠ only when he has A♠;
- most moves introduce new information. Example: the opponent led A♠. Now, all the worlds in which he hadn't A♠ can be regarded as impossible;
- payoff values for a leaf l are defined only in these worlds in which we can access the leaf. In fact, the players know the situation (more or less) after the last move in many games. Thus – for a particular leaf – payoffs in most worlds make no sense.

To incorporate this perspective, the following assumptions can be made:

- payoff vector is a partial function – $payoff(l) : \Omega \rightharpoonup R$ (values for some l, w can be undefined: $payoff(l)[w] = undef$). It's good to assume that: $a + undef = a \cdot undef = max(a, undef) = min(a, undef) = a$;
- legal moves are determined by a function Acc. $Acc(s)$ denotes the set of worlds in which node s is accessible. Acc can be implemented as follows:
  1. if s is a leaf then $Acc(s) = \{w \in \Omega : payoff(s)[w] \neq undef\}$,
  2. otherwise $Acc(s) = \bigcup_{s' \in Succ(s)} Acc(s')$.
- every move can reveal some new information, so the beliefs may change as the state changes – $\Omega_{MAX}, \Omega_{MIN} : State \times \Omega \rightarrow \mathcal{P}(\Omega)$.

The resulting structure resembles in a way the semantics underpinning $\mathcal{LORA}$, a complex modal logic for BDI agents proposed in (Wooldridge 2000) – with the game tree defining the branching of time, and $\Omega_{MAX}, \Omega_{MIN}$ standing for the belief accessibility relations – although $\mathcal{LORA}$ proceeds with qualitative, not probabilistic approach to beliefs.

Next generalization:

gvm (Game, s, player, belief);

Generalized vector minimaxing. Returns the evaluation vector $(eval[w_1], ..., eval[w_n])$ for node s, together with the player's chosen move. Parameters:

Game: game definition, including functions $Succ : State \to \mathcal{P}(State)$ – returning the set of successors for each node, $payoff : State \times \Omega \to \mathbb{R}$ – returning MAX's payoffs, Bel_{MIN},$Bel_{MAX} : State \times \Omega \to ((\Omega \to [0,1]) \to [0,1])$ – beliefs about the likelihood of particular opponent's beliefs in a particular situation;

$s : State$ – the game state being considered;

$player : \{MAX, MIN\}$ – the agent who makes the decision at the state;

$belief : \Omega \to [0,1]$ – the agent's actual belief (a probability function);

if $Succ(s) = \emptyset$ then return $(nil, payoff(s))$;
else:

- for every $s' \in Succ(s)$, $w \in \Omega$, and for every possible opponent's belief *oblf* simulate the opponent's minimaxing:
$$opp_{s'}[oblf, w] = \begin{cases} gvm(Game, s', \text{MIN}, oblf)[w] & \text{if } player = \text{MAX} \\ gvm(Game, s', \text{MAX}, oblf)[w] & \text{if } player = \text{MIN} \end{cases}$$
- compute the expected payoff for every $s' \in Succ(s)$, $w \in \Omega$:
$$e_{s'}[w] = \begin{cases} \sum_{oblf} Bel_{MIN}(s, w, oblf) \cdot opp_{s'}[oblf, w] & \text{if } player = \text{MAX} \\ \sum_{oblf} Bel_{MAX}(s, w, oblf) \cdot opp_{s'}[oblf, w] & \text{if } player = \text{MIN} \end{cases}$$
- if *player*=MAX then return $(s', e_{s'})$ such that $\sum_{w \in \Omega} belief(w) \cdot e_{s'}[w]$ is maximal.
else return $(s', e_{s'})$ such that $\sum_{w \in \Omega} belief(w) \cdot e_{s'}[w]$ is minimal.

Fig. 10. Generalized vector minimaxing revisited.

– players may be able to determine some probabilities for the possible worlds – not only to tell which worlds are plausible and which implausible now. Thus, an actual belief may be a probability function $[0,1] \to \Omega$ instead of being just a subset of Ω;
– in a given state of a game (and a world), more than 1 belief may be possible within the opponent's model. This would mean that the player doesn't know the opponent's reasoning scheme precisely and is bound to guess which belief states can result from the opponent's information analysis. He may also consider some of the possible opponent's beliefs more likely than the others.

A new version of *gvm* that takes the new possibilities into account is shown on figure 10; *findoptimal* algorithm can be generalized in a similar way.

The set of all the possible beliefs is in general infinite, so it demands for some reduction of the problem. A clever sampling of the set may be a good solution.

3.5 Dealing with More Capable Opponents

In the actual experiments, functions like Ω_{MIN} were designed to describe what a rational opponent *must* know in a given situation. On the other hand, the

opponent *may* know (or believe he knows) much more. He can even happen to know the whole situation – he may have guessed it from his own card, the MAX player's first move, or even kibitzers' facial expressions. Frank's best defense assumptions refer clearly to what the opponent can know in the worst possible case.

To deal with such 'possibly more capable' opponent, the player should consider every possible opponent's belief from between what he must and what he can know. For the experiment setting (no difference in probability of plausible worlds) this would mean maximizing the expected value over all the belief sets B such that $\{w\} \subseteq B(s,w) \subseteq \Omega_{MIN}(s,w)$. Thus, to model this situation accurately, it is sufficient to assume

$$Bel_{MIN} = \begin{cases} \frac{1}{k} & \textbf{if} \quad \exists_B \ \{w\} \subseteq B(s,w) \subseteq \Omega_{MIN}(s,w) \ \wedge \ p(w) = \begin{cases} \frac{1}{|B|} & \text{if } w \in B \\ 0 & \text{else} \end{cases} \\ 0 & \textbf{otherwise} \end{cases}$$

within the input for the generalized version of *gvm* or *findoptimal*. [4]

In the general case (analyzed in the previous section) we can have beliefs about opponent's beliefs defined explicitly with probability functions Bel_{MIN}, Bel_{MAX}. Simply, when we suspect the opponent of being more capable than just looking at his card and/or the board, it's good to design the functions so that if any opponent's belief is assumed possible then all the more precise beliefs are also assumed possible.

4 Conclusions

This paper advocates a thesis that assuming a complete omniscience of the opponent may be not quite reasonable in games with incomplete information. Instead, the player should optimize his strategy against the expected performance of the other agent (in the mathematical sense). If the player can identify the opponent's belief for various possible situations, he can do some reasoning in the way Gambäck, Rayner and Pell showed for the specific case of Bridge bidding. Algorithms: *gvm* and *findoptimal* implement the idea, and the results of the experiments suggest that the 'reasonably good defense model', proposed in this paper, may make sense after all. Of course, the algorithms – especially *findoptimal* – are too inefficient to be used in practice, but they can provide a good benchmark for evaluation of suboptimal, faster ones.

The defense model proposed here – in contrast to the model by Frank and Basin – emphasizes the importance of a good information-processing subsystem, necessary to acquire and maintain an adequate knowledge about the opponent. The actual opponent model may be derived from the game rules or learned by the playing agent during the play. The point is that if the player has any (even uncertain) information about the agent he plays against available, he should use it instead of ignoring it. And if the player has *really* no information about the other agent, he may be better off assuming average capabilities of the opponent, rather than capabilities the opponent is unlikely to possess.

[4] k must represent the number of possible B sets in the equation to keep the probability function normalized.

References

1. Carmel D., Markovitch S. (1996), Learning and Using Opponent Models in Adversary Search, Technical Report CIS9609, Technion, Haifa.
2. Corlett R., Todd S. (1985), A Monte-Carlo Approach to Uncertain Inference, In Ross P., ed., *Proceedings of the Conference on Artificial Intelligence and Simulation of Behaviour*, pp. 28-38.
3. Frank I. (1996), *Search and Planning under Incomplete Information. A Study using Bridge Card Play*, Ph.D. diss., University of Edinburgh. Published in 1998 by Springer-Verlag.
4. Frank I., Basin D. (1998a), Search in Games with Incomplete Information. A Case Study using Bridge Card Play, *Artificial Intelligence*, 100(1-2), pp. 87-123.
5. Frank I., Basin D. (1998b), Optimal Play Against Best Defense: Complexity and Heuristics, In Proceedings of the First International Conference on Computers and Games, *Lecture Notes in Computer Science*, Vol. 1558, Springer–Verlag.
6. Frank I., Basin D., Matsubara H. (1998), Finding Optimal Strategies for Imperfect Information Games, In *Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI-98*.
7. Gambäck B., Rayner M., Pell B. (1993), Pragmatic Reasoning in Bridge, Technical Report No. 299, University of Cambridge.
8. Ginsberg M. (1999), GIB: Steps Toward an Expert-Level Bridge-Playing Program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pp. 584-589.
9. Ginsberg M. (1996), How Computers Will Play Bridge. *The Bridge World*, June.
10. Klir G.J., Folger T.A. (1988), *Fuzzy Sets, Uncertainty and Information*, Prentice Hall, Englewood Cliffs.
11. Kofler E. (1963) *Wstęp do teorii gier. Zarys popularny* [*An Introduction to Game Theory. Popular Approach*]. PZWS, Warszawa.
12. Sen S., Weiss G. (1999), Learning in Multiagent Systems. In: Weiss G. (ed.), *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, pp. 259-298, MIT Press, Cambridge, Mass.
13. Weiss G., ed. (1999), *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*, MIT Press, Cambridge, Mass.
14. Wooldridge M. (2000), *Reasoning about Rational Agents*, MIT Press, Cambridge, Mass.

Towards Inferring Labelling Heuristics for CSP Application Domains

Zeynep Kızıltan, Pierre Flener, and Brahim Hnich

Computer Science Division, Department of Information Science
Uppsala University, Box 513, S – 751 20 Uppsala, Sweden
{Zeynep.Kiziltan, Pierre.Flener, Brahim.Hnich}@dis.uu.se

Abstract. Many real-life problems can be represented as constraint satisfaction problems (CSPs) and then be solved using constraint solvers, in which labelling heuristics are used to fine-tune the performance of the underlying search algorithm. However, few guidelines have been proposed for the application domains of these heuristics. If a mapping between application domains and heuristics is known to the solver, then modellers can — if they wish so — be relieved from figuring out which heuristic to indicate or implement. Instead of inferring the application domains of (known) heuristics, we advocate inferring (known or new) heuristics for application domains. Our approach is to first formalise a CSP application domain as a family of models, so as to exhibit the generic constraint store for all models in that family. Second, family-specific labelling heuristics are inferred by analysing the interaction of a given search algorithm with this generic constraint store. We illustrate our approach on a domain of subset problems.

1 Introduction

Many real-life problems are *constraint satisfaction problems* (CSPs), where appropriate values for the variables of the problem have to be found within their domains, subject to some constraints. Examples are production planning subject to demand and resource availability, air traffic control subject to safety protocols, etc. Many of these problems can be programmed as constraint models and then be solved using constraint solvers, such as CLP(FD) [2] and OPL [17].

Constraint solvers are equipped with a *search algorithm*, such as forward-checking, and *labelling heuristics*, one of which is the default. To enhance the performance of constraint models, a lot of research has been made in recent years to develop new labelling heuristics, which concern the choice of the next variable to branch on during the search and the choice of the value to be assigned to that variable. These heuristics significantly reduce the search space [15].

However, little is said about the application domains of these heuristics, so modellers find it difficult to decide when to apply a particular heuristic, and when not. Indeed, there is no universally best heuristic for all instances of all constraint models (see, e.g., [16]), unless NP=P. Thus, we are only told that a particular heuristic was "best" for the particular instances used to carry out

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 275–289, 2001.

some experiments with some particular models. Therefore, the performance of heuristics is not only model-dependent but also instance-dependent, i.e., for a given constraint model, a heuristic can perform well for some (distributions on the) instances, but very poorly on others; this is taken into account by some generators of model-specific solvers [3,9,12].

Instead of inferring the application domains of (known) heuristics, we advocate inferring (known or new) heuristics for application domains. Obviously, the "smaller" an application domain, the "better" its inferrable heuristics. Our two-step approach is to first formalise an application domain as a family of CSP models, so as to exhibit the generic constraint store for all models in that family. Second, the interaction — for a given search algorithm — between the constraints in this generic store and the domain propagation during search is examined, so as to infer suitable heuristics for any model in that family. Due to the instance sensitivity of heuristics, the outcome of this process usually is a *set* of heuristics, rather than a single one. In this paper, we illustrate this approach on a domain of subset problems.

If a mapping between application domains and heuristics is known to the solver, then modellers can — if they wish so — be relieved from the procedural aspect of modelling, namely figuring out which heuristic to indicate or implement. Forcing modellers to deal with this procedural aspect may not only add a challenging step but also has the disadvantage that they must commit — at modelling time — to a *single* heuristic and thus expose their models to the instance sensitivity of heuristics. In companion work [7,11], we address the issue of selecting or switching — at solving time — among the inferred family-specific heuristics resulting from our approach, according to the instance to be solved. Our ultimate aim is thus a new generation of more intelligent solvers that allow CSP modellers to concentrate on the declarative aspect of modelling, without compromising (much) on efficiency.

This paper is organised as follows. In Section 2, we introduce the notion of family of CSP models as a formalisation of an application domain. We illustrate this with a domain of subset problems and exhibit a generic finite-domain constraint store of a family for this domain. Then, in Section 3, we present our analysis of this generic constraint store, infer two labelling heuristics, and show our initial empirical results. Finally, in Section 4, we conclude, compare with related work, and discuss directions for future research.

2 CSP Model Families

Informally, an *application domain* is a set of "related" CSPs. For instance, in the *SUBSET* domain, a given number of elements have to be selected from a given finite set such that any two of them satisfy some constraint p. In this domain, CSPs are related in the sense that the actual constraint p differs between them. Sample CSPs in this domain are finding a clique of a given size within a given graph (where p requires that any two vertices of the clique be connected by an edge of the graph) and finding an independent subset of a given size among

the vertices of a given graph (where p says that any two vertices of the subet must not be connected by an edge of the graph). Application domains of coarser granularity are scheduling, configuration, resource allocation, and so on.

For a given constraint modelling language, a *CSP model family* is an open CSP model in that language, '*open*' in the sense that some of its (predicate or type) symbols are neither primitive to the language nor defined in the model. An actual *CSP model* is *closed*, in the sense that all its symbols must be primitive or defined. From a model family, a model can thus be obtained by substituting closed types and closed formulas for all its open symbols, and possibly by adding parameters. Model families can be used to formalise application domains. There are in general several ways of formalising a domain as a model family, in a given language, namely depending on the chosen data modelling. An *instance* of a CSP model M is obtained from M by replacing all its formal input parameters by actual values and dropping the universal quantifications on these parameters. An instance of a model is thus also a model, albeit without input parameters.

Example 2.1. Assume CSP models are written in a very expressive, purely declarative, typed, set-oriented, first-order logic constraint modelling language, such as our ESRA [6,4], which is designed to be higher-level than even OPL [17]. (We can automatically compile [5] ESRA programs into lower-level languages such as OPL.) Since ESRA has set variables (unlike OPL), the following (sugared version of an) ESRA model family is a candidate formalisation of the *SUBSET* domain:

$$\begin{array}{c}\forall T, S : set(\alpha)\,.\,\forall k : int\,.\,(subset(T^+, k^+, S) \leftrightarrow S \subseteq T \wedge size(S, k) \wedge \\ \forall t_i, t_j : \alpha\,.\,(t_i \in S \wedge t_j \in S \wedge t_i \mathrel{/=} t_j \rightarrow p(t_i, t_j)))\end{array} \qquad (Subset)$$

where the superscript $^+$ designates the input parameters. In words, sets S and T of elements of type α are in the *subset*/3 relation with integer k iff S is a set of k elements from T, such that any two distinct elements t_i and t_j of S satisfy constraint p. The only open symbols are type α and constraint p, as $size$, $\subseteq$, $\in$, and $/=$ are primitives of ESRA, with the usual meanings. From the *Subset* model family, we can obtain the following (sugared) ESRA model:

$$\begin{array}{c}\forall V, C : set(int)\,.\,\forall k : int\,.\,\forall E : set(int \times int)\,. \\ (clique_k(\langle V^+, E^+\rangle, k^+, C) \leftrightarrow C \subseteq V \wedge size(C, k) \wedge \\ \forall v_i, v_j : int\,.\,(v_i \in C \wedge v_j \in C \wedge v_i \mathrel{/=} v_j \rightarrow \langle v_i, v_j\rangle \in E))\end{array} \qquad (clique_k)$$

It is a model for finding a clique C of an undirected graph (given through its integer vertex set V and edge set E), such that the clique has k vertices.

Example 2.2. At a lower level of expressiveness, say when set variables are not available (such as in CLP(FD) [2] and OPL [17]), the usual representation of an unknown subset S of a given *finite* set T (of n elements) is a mapping from T into Boolean variables (in $\{0, 1\}$), that is one conceptually maintains n couples $\langle t_i, B_i\rangle$ where the (initially non-ground) Boolean B_i expresses whether the

(always ground) element t_i of T is a member of S or not:[1]

$$\forall t_i : \alpha \,.\, t_i \in T \rightarrow (B_i \leftrightarrow t_i \in S) \tag{1}$$

This Boolean representation of set variables consumes more memory than the set-interval representation of CONJUNTO [8] and OZ [13], but both have been shown to create the same $O(2^n)$ search space [8].

Given this Boolean representation of the sought subset S, restricting its size to k can be expressed as the following n-ary constraint:

$$\sum_{i=1}^{n} B_i = k \tag{2}$$

Let us also look at the remaining part of *SUBSET*, which requires that any two distinct elements of the subset S of T must satisfy a constraint p. Formally (using the sugared ESRA syntax again, for the sake of symbolic reasoning):

$$S \subseteq T \wedge \forall t_i, t_j : \alpha \,.\, t_i \in S \wedge t_j \in S \wedge t_i \not= t_j \rightarrow p(t_i, t_j)$$

This implies

$$\forall t_i, t_j : \alpha \,.\, t_i \in T \wedge t_j \in T \wedge t_i \in S \wedge t_j \in S \wedge t_i \not= t_j \rightarrow p(t_i, t_j)$$

which is equivalent to

$$\forall t_i, t_j : \alpha \,.\, t_i \in T \wedge t_j \in T \wedge t_i \not= t_j \wedge \neg p(t_i, t_j) \rightarrow \neg(t_i \in S \wedge t_j \in S)$$

By (1), this can be rewritten as

$$\forall t_i, t_j : \alpha \,.\, t_i \in T \wedge t_j \in T \wedge t_i \not= t_j \wedge \neg p(t_i, t_j) \rightarrow \neg(B_i \wedge B_j) \tag{3}$$

The sugared version of an OPL/CLP(FD) model family formalising the *SUBSET* domain thus consists of constraints (2) and (3); we denote it by $Subset_B$. For any two distinct elements t_i and t_j of the given set T, with Boolean variables B_i and B_j, if $p(t_i, t_j)$ does not hold, the following binary constraint arises:

$$\neg(B_i \wedge B_j) \tag{4}$$

It is crucial to note that the actual finite-domain constraints are thus *not* in terms of p, hence p can be *any* formula. Therefore, the generic finite-domain constraint store for *any* instance of *any* model of the $Subset_B$ family is over a set of (only) Boolean variables. It contains an instance-dependent number of binary constraints of the form (4), as well as the (always unique) n-ary constraint (2).

As the set-interval representation of set variables does not allow the definition of some (to us) desirable high-level primitives, such as universal quantification over elements of non-ground sets, the set variables of ESRA (see Example 2.1) are compiled [5,6] using the Boolean representation of Example 2.2. In the remainder of this paper, our approach to inferring labelling heuristics from an application domain is illustrated on the *SUBSET* domain, and we (thus) focus on its Boolean modelling in the $Subset_B$ family.

[1] In formulas, we use atom B_i as an abbreviation for $B_i = 1$.

3 Inferring Labelling Heuristics

It is known that the order in which the variables are considered for instantiation, and the order in which the values are attempted for assignment to variables during search have a substantial impact on the number of backtracks performed and the time taken by a search algorithm to solve a CSP model. Deciding on these orders is the objective of labelling heuristics. We now infer some labelling heuristics for the *SUBSET* domain by examining the domain propagation performed on the generic constraint store — for the $Subset_B$ family — by a search algorithm during labelling. For the sake of illustration, we here choose the forward checking (FC) algorithm, which is used in many solvers. It works as follows: Whenever a variable is labelled by a value v, the values of the future variables that are inconsistent with v are removed from the domains of these variables.

In Section 3.1, we present our analysis of the obtained generic constraint store. Next, in Section 3.2, we infer some FC labelling heuristics for $Subset_B$ models. Finally, in Section 3.3, we report on our initial experimental results.

3.1 Analysis of the Generic Constraint Store

We analyse the generic constraint store using the values n (the size of the given set T, hence the number of Boolean variables involved) and k (the given size of the sought subset S). In models of the $Subset_B$ family, each Boolean variable B_i in $\{B_1, \ldots, B_n\}$ is at any moment associated with the set V_i of still unassigned variables B_j (where $1 \leq j \leq n$) that constrain B_i with a binary constraint of the form (4). A binary constraint of this form requires that the variables B_i and B_j cannot simultaneously be assigned 1. Furthermore, the n-ary constraint (2) restricts all the variables such that k of them must be assigned 1. Let k_0 (resp. k_1) be the *current* number of variables that have yet to be assigned 0 (resp. 1). Initially (before the labelling), $k_0 = n-k$ and $k_1 = k$. During labelling, the values of k_0 and k_1 decrease because of the assignments and propagation. If either k_0 or k_1 reaches 0, the propagation caused by the n-ary constraint forces the other one to also become 0. Therefore, at the end (after the labelling), $k_0 = k_1 = 0$. Note that the mathematical variables $V_1 \ldots V_n$, k_0, k_1 are only explanatory devices, but not actually stored and manipulated anywhere.

We now monitor the FC propagations triggered by the assignment of values (from $\{0, 1\}$) to the Boolean variables. The ordering of the variables and values is irrelevant in this analysis: suitable labelling heuristics will be inferred in Section 3.2. When $k_0 > 0$ and $k_1 > 0$, we consider two cases, namely Case A, the assignment of 0, and Case B, the assignment of 1 to the chosen variable, say B_i.

Case A. If B_i is assigned 0, the current number of variables that have yet to be assigned 0 is decremented by 1, so k_0 becomes $k_0 - 1$. Two sub-cases arise now:

- If $k_0 = 0$ now, then all the k_1 yet unassigned variables are assigned 1 during propagation due to (only) the n-ary constraint (2), leading to $k_1 = 0$ also. Now exactly $n - k$ variables have been assigned 0 and k variables have been assigned 1. However, if there is a binary constraint of the form (4) between

any two of these k_1 variables, then this assignment fails, which leads to backtracking. Otherwise, this assignment succeeds.
- If $k_0 > 0$ still, then, for all $v \in V_i$, the domain of v remains the same, because the assignment of 0 to any variable in a binary constraint of the form (4) always succeeds without propagation.

By the instantiation of a variable by 0, there is thus a possibility of backtracking only if k_0 reaches 0, because the assignment may fail.

Case B. If B_i is assigned 1, the current number of variables that have yet to be assigned 1 is decremented by 1, so k_1 becomes $k_1 - 1$. Two sub-cases arise now:

- If $k_1 = 0$ now, then all the k_0 yet unassigned variables are assigned 0 during propagation due to (only) the n-ary constraint (2), leading to $k_0 = 0$ also. Now exactly k variables have been assigned 1 and $n - k$ variables have been assigned 0, without violating any constraints. Indeed, as seen in Case A, the assignment of 0 to a variable fails only if k_0 becomes 0 *and* there is a binary constraint between any two of the k_1 variables. However, there are here *no* unassigned variables left, as $k_1 = 0$ already. Therefore, this assignment always succeeds.
- If $k_1 > 0$ still, then, for all $v \in V_i$, the variable v is assigned 0 during propagation because of the binary constraints of the form (4). Thus, k_0 becomes $k_0 - |V_i|$. The new value of k_0 now gives rise to the following sub-sub-case analysis:
  - If $k_0 < 0$ now, then one of these assignments must fail and immediate backtracking occurs.
  - If $k_0 = 0$ now, then all the k_1 yet unassigned variables are assigned 1 during propagation, leading to $k_1 = 0$ also. As seen in Case A, if there is a binary constraint of the form (4) between any two of these k_1 variables, then this assignment fails, which leads to backtracking. Otherwise, this assignment succeeds.
  - If $k_0 > 0$ now, then this assignment succeeds.

By the instantiation of a variable by 1, there is thus a possibility of backtracking only if k_0 reaches 0 first. Should k_0 become negative, the assignment fails, and thus an immediate backtracking occurs. On the other hand, the assignment always succeeds if k_1 reaches 0 first.

It is *very* important to notice that Case B may include Case A. On the other hand, Case A never includes the general situation of Case B. Therefore, the analysis became of *finite* size and complete, as there is no case where it is impossible to exactly foretell *all* propagations!

3.2 Inference of Heuristics

In models of the $Subset_B$ family, the assignment — under FC search — of 0 to a Boolean variable triggers propagation *only* when k_0 reaches 0, and this *independently* of the order of the variables being instantiated by 0 so far. Therefore, if the *set* of variables that will be assigned 0 is not chosen carefully (e.g., when

there are no binary constraints between them, in which case there probably are binary constraints between the other variables), backtracking is unavoidable once k_0 reaches 0. The only way to avoid backtracking is to choose the right set of $n - k$ variables that are assigned 0. However, finding such a subset of the Boolean variables is itself a subset problem.

The assignment of 1 to a variable is noteworthy because *every* assignment caused by propagation upon $k_1 = 0$ succeeds, so that no backtracking can happen. Also, the order of the variables being assigned 1 is quite important because it can significantly affect the decrease in k_0. Indeed, as seen in Case B, if $k_1 > 0$ still, then k_0 becomes $k_0 - |V_i|$. The variable B_i being assigned 1 is associated with a set V_i (the set of the still unassigned variables that constrain B_i) that thus directly affects the decrement in k_0. If the variables being assigned 1 are ordered in a way that they do not cause much decrease in k_0, then backtracking when $k_0 < 0$ and any possible backtracking when $k_0 = 0$ are delayed. Backtrack-free assignment is thus guaranteed by allowing k_1 to reach 0 first. However, backtrack-free assignment is not guaranteed if it is k_0 that reaches 0 first.

We can thus infer the following two labelling heuristics from the previous considerations:

- *If there is at least one solution*, we should instantiate some variables by 1, and try to keep each $|V_i|$ as small as possible if we want k_1 to reach 0 first (which leads to backtrack-free assignment). Thus, during FC search, if we choose a variable that is participating in the *smallest* number of binary constraints, then we force k_1 to become 0 before (or at the same time) as k_0 does, because, by this way, we achieve a small decrease in k_0. This heuristic can be seen as an instance of the succeed-first principle.
- *If there is no solution*, then it is impossible to reach the state $k_1 = 0$. Search effort can then be saved by forcing the search to reach a state with definite backtracking (when $k_0 < 0$) or possible backtracking (when $k_0 = 0$) as soon as possible. Thus, during FC search, if we choose a variable that is participating in the *largest* number of binary constraints, then we force k_0 to be negative or to become 0 before k_1 does, because, by this way, we achieve a big decrease in k_0. The value ordering is thus irrelevant. This heuristic can be seen as an instance of the fail-first principle.

As it is initially unknown whether there is a solution or not, it is very difficult to choose which of these two heuristics to use in order to guide the search process. This paper is only concerned with the inference of heuristics; the issue of deciding when to use which one, or when to switch between them, is addressed in companion work [7,11].

Following these considerations, we implemented the following static labelling heuristics, namely in SICSTUS CLP(FD) (which has an FC solver):

- H_s^1, which chooses the variable that is constraining the smallest number of variables, and assigns the value 1 first.
- H_l^0 (resp. H_l^1), which chooses the variable that is constraining the largest number of variables, and assigns the value 0 (resp. 1) first.

Being static, these labelling heuristics choose a variable that is *initially* constraining the smallest/largest number of variables. Note that this implementation of the heuristics is our choice, but that the heuristics could be implemented in another way, say by re-ordering the variables at solving-time. Investigation of the superiority or the inferiority of such dynamic variable orderings, which choose a variable that is constraining the smallest/largest number of the future (yet unassigned) variables, to the static ones is left as future work.

3.3 Experiments with the Heuristics

Experimental Setting. We measured the cost (in CPU time and in number of backtracks) of our heuristics on a very large number of instances of the models of the $Subset_B$ family. These experiments confirmed the anticipated strengths and weaknesses of the heuristics, which are exploited in our companion work on deciding when to use which heuristic, or when to switch between them [7,11].

For binary CSPs, a class[2] of instances is usually characterised by a tuple $\langle n, m, p_1, p_2 \rangle$, where n is the number of variables, m is the (assumed constant) domain size for all variables, p_1 is the (assumed constant) constraint density, and p_2 is the (assumed constant) tightness of the individual constraints. Experiments are then conducted by iterating over an interval of instance classes and generating a suitably sized sample of random instances for each class. For each sample, the median or average solving cost is computed.

However, our generic constraint store features a non-binary constraint, so we cannot literally apply this characterisation of instance classes. In any case, the latter has been criticised [1] because it is unrealistic to have a *constant* tightness p_2 for all constraints, so that many possible instances can never be generated. For these two reasons, we developed the following characterisation of instance classes, which is specific to the considered family. It is not subject to any of the criticisms in [1], because it exploits the structure of the generic constraint store.

The generic finite-domain constraint store for the $Subset_B$ family is parameterised by the number n of Boolean variables involved (i.e., the size of the given set T) and the given size k of the sought subset S, and contains an instance-dependent number b of binary constraints of the form (4). The number n of variables and the density p_1 of the constraints are kept from the previous characterisation, with p_1 being $\frac{b}{n(n-1)/2}$ here. The domain size m is dropped, as it always is 2, because we need only consider the Boolean domain $\{0, 1\}$. Since the considered binary constraints are of the form $\neg(B_i \wedge B_j)$, their tightness always is 3/4 and thus does not become a parameter. The tightness of the n-ary constraint however is $\binom{n}{k}/2^n$, and thus varies with n and k. As we already use n, the size k becomes the final parameter in our characterisation of instance classes, which is thus summarised by the triple $\langle n, p_1, k \rangle$.

For the purpose of this paper, we generated random instances in a coarse way, by not considering all possible values of n up to a given limit. The number n of variables ranged over the interval 10..120, by increments of 10. We varied

[2] A class (of instances) is not to be confused with a family (of CSP models).

the density p_1 over the interval 0.1..1, by increments of 0.1. The values of k ranged over the interval $1..n$, by increments of 1. Considering the sizes of these intervals, the number of our experiments was huge and their execution was very time-consuming. Given more time, instances generated in a more fine-grained way could be used instead and help to make our (future) results more precise. Our objective here only is to show the heuristics in action, but not to provide the most detailed statistics for our companion work on deciding when to use which heuristic, or when to switch between them [7,11].

Rather than only comparing the inferred heuristics to each other, we also compared them to some others. For time reasons, we restricted ourselves to the following two additional heuristics:

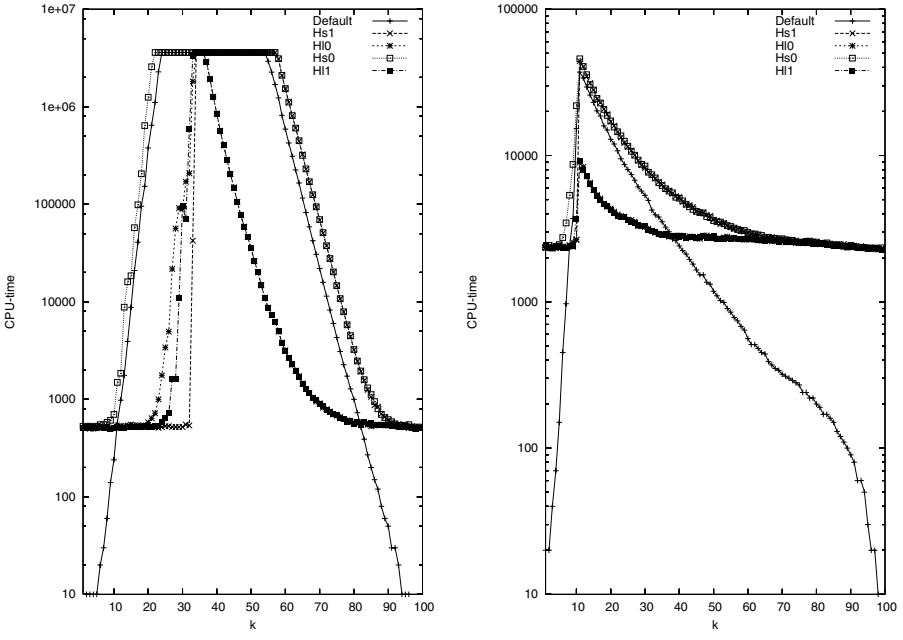
- H_s^0, which chooses the variable that is constraining the smallest number of variables, and assigns the value 0 first.
- $Default$, the default labelling heuristic of SICSTUS CLP(FD), which labels the leftmost variable in the provided sequence of variables, and the domain of the chosen variable is explored in ascending order (i.e., 0 first in our case).

The heuristic H_s^0 is a natural complement to the inferred heuristics, and was also implemented in SICSTUS CLP(FD). In the absence of a labelling heuristic provided by the modeller, each solver uses its default heuristic. Since our experiments were conducted in SICSTUS CLP(FD), its default heuristic *had to* be used here. (The experiments thus have to be repeated for each FC solver, because their default heuristics change.)

If a combination of the inferred heuristics beats — on the average over numerous instances of the family — the default heuristic of the solver, then this combination can become a family-specific and even highly instance-sensitive default heuristic of the solver. The determination of such a combination is addressed in our companion work [7,11]. If this idea is repeated for other families, then the modellers can — if they wish so — be relieved from the procedural aspect of modelling and even be protected from the instance sensitivity of their heuristics.

Our experiments were made over *random* instances (of models) of the considered family for the following reason. Towards using *real-life* instances, we would have had to first pick some models within the considered family, but we would then have been unable to justify why these models were picked rather than some others. The purpose of our experiments [10] was to generate statistics that guide us in our companion work [7,11], where we aim at a family-specific default heuristic for a *solver*, which must be able to handle random instances over that entire family. We do not aim at a heuristic for a specific *model*, which would have to be able to handle (only) real-life instances of (only) that model.

Experiments. Having thus chosen the intervals and increments for the parameters in our characterisation of an instance class, we randomly generated many different instances and then used the 5 heuristics in order to solve them or prove that they have no solutions. Some of the instances were obviously too difficult to solve or disprove within a reasonable amount of time. Consequently, to save time in our experiments, we used a time-out (of 3,600,000ms) on the CPU time; upon time-out, the current number of backtracks was recorded.



(a) $p_1 = 0.1$ (b) $p_1 = 0.5$

Fig. 1. CPU-time (in ms) in terms of k for the 5 heuristics on $n = 100$

In order to analyse the effects of each heuristic on different instances, we drew various charts, for example by keeping n and p_1 constant and plotting the median costs of the samples for each k. Figure 1 shows an example of the behaviours of the 5 heuristics in terms of CPU-time on the instances where $n = 100$, with $p_1 = 0.1$ and $p_1 = 0.5$, respectively. Figure 2 shows their behaviours in terms of the number of backtracks on the same instances.

These figures do not show that the generated instances exhibit three very interesting regions in terms of k, no matter what n and p_1 are: up to some value v of k, all instances have a solution; then, until some other value w of k, some instances have a solution and some do not; beyond w, all instances have no solution. A visible interesting observation is that, without a time-out, the solving-times for instances increase with k until some point, whereupon they decrease. With the heuristics we used, we recorded time-outs in all three of the mentioned regions. After taking the median cost of the generated sample of random instances for each class $\langle n, p_1, k\rangle$, we observed three different zones in terms of k: up to some value j in $0..n$, the instance with the median cost has a solution; from some other value l in $j + 1 \,..\, n + 1$, the instance with the median cost has no solution; in-between, the instance with the median cost timed out. It is in general unknown where j and l are compared to v and w. The values of j, l, v, w depend on n and p_1.

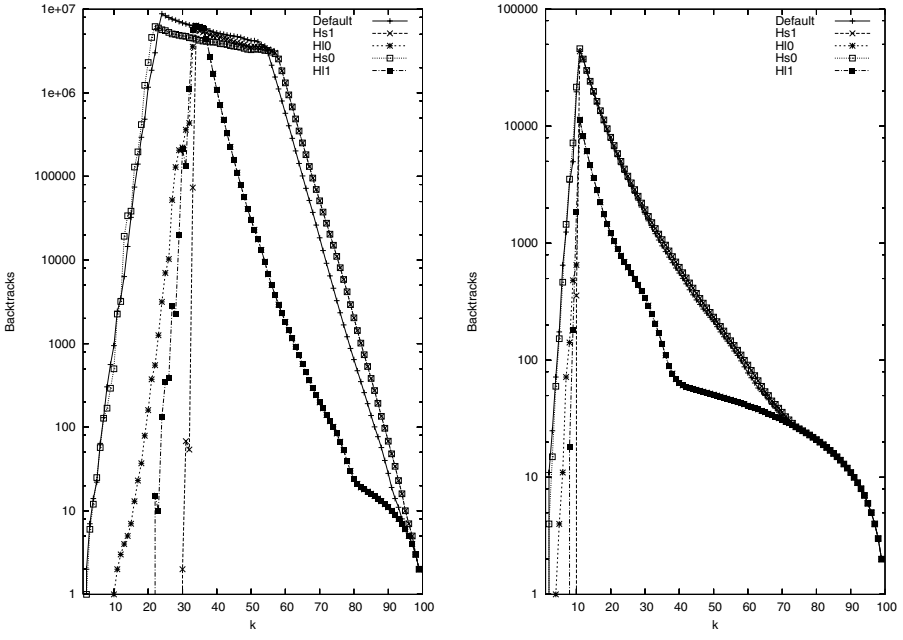
The position of k relative j and l yields the following analysis of the behaviours of the heuristics in terms of the CPU-time they take (see Figure 1):

- Over $1..j$, the heuristic H_s^1 always finds a solution, in mostly *constant* CPU-time. *Default* performs the best until k reaches some d in $0..j$, where d is small. However, over $d+1$ .. j, the heuristic H_s^1 outperforms $Default$. The heuristics H_l^0 and H_l^1 perform as well as H_s^1 until k reaches some i in $1..j$. However, over $i+1$ .. j, the heuristic H_s^1 outperforms H_l^0 and H_l^1. Heuristic H_s^0 usually has the worst performance. In conclusion, over $1..j$, the heuristic H_s^1 is the *best* over $d+1$ .. j, with $1..d$ being always a very small interval. The range of k where H_s^1 performs the best varies in size with respect to p_1, given n: compare Figures 1(a) and 1(b).
- Over $j+1$ .. $l-1$, we cannot compare the heuristics because they all timed out. This can be observed in Figure 1(a) for k in 34..37.
- Over $l..n$, the heuristic H_s^1 always proves that there is no solution, in decreasing CPU-time. Heuristic H_s^0 usually has the worst performance. In this range, the heuristic H_s^1 is always outperformed by H_l^0 and H_l^1, and performs as badly as H_s^0. The heuristics H_l^0 and H_l^1 perform the *best* until k reaches some i in $l..n$, whereupon *Default* outperforms all the others. The range of k where H_l^0 and H_l^1, or *Default* perform the best varies in size with respect to p_1, given n: compare Figures 1(a) and 1(b).

The heuristic H_s^1 mostly performs the best when there is an observed solution. This can easily be explained by the fact that it was designed to try and find a solution, while assuming there is one. The heuristics H_l^0 and H_l^1 mostly perform the best when there is no observed solution. This is because they were designed to prove that there is no solution, while assuming there is none. The reason why $Default$ sometimes outperforms the other 4 heuristics is that it has no solving-time overhead. Somewhere in $j+1$ .. $l-1$, a phase transition from the soluble region to the non-soluble region occurs, and all the heuristics failed to efficiently handle these instances and thus timed out.

The position of k relative j and l yields an analysis of the behaviours of the heuristics in terms of the number of backtracks they make (see Figure 2):

- Over $1..j$, the heuristic H_s^1 always finds a solution, mostly in 0 backtracks. *Default* always performs worse than H_s^1. The heuristics H_l^0 and H_l^1 initially perform as well as H_s^1, but start backtracking earlier. Heuristic H_s^0 usually has the worst performance. In conclusion, over $1..j$, the heuristic H_s^1 is always the *best*. The range of k where H_s^1 performs 0 backtracks varies in size with respect to p_1, given n: compare Figures 2(a) and 2(b).
- Over $j+1$ .. $l-1$, we cannot compare the heuristics because they all timed out. This can be observed in Figure 2(a) for k in 34..37.
- Over $l..n$, the heuristic H_s^1 always proves that there is no solution, in decreasing numbers of backtracks. Heuristic H_s^0 usually has the worst performance. In this range, the heuristic H_s^1 is always outperformed by H_l^0 and H_l^1, and performs as badly as H_s^0. The heuristics H_l^0 and H_l^1 perform the *best* until k reaches some i in $l..n$, whereupon all the 5 heuristics perform the same



(a) $p_1 = 0.1$ (b) $p_1 = 0.5$

Fig. 2. Number of backtracks in terms of k for the 5 heuristics on $n = 100$

number of backtracks. The range of k where H_l^0 and H_l^1 (resp. all the 5 heuristics) perform the best (resp. the same) varies in size with respect to p_1, given n: compare Figures 2(a) and 2(b).

The heuristic H_s^1 always performs the best in number of backtracks (and mostly with 0 backtracks) when there is an observed solution, because it was designed to try and find a solution, while assuming there is one. The heuristics H_l^0 and H_l^1 mostly perform the best in number of backtracks when there is no observed solution. This is because they were designed to prove that there is no solution, while assuming there is none. Somewhere in $j+1 \,..\, l-1$, a phase transition from the soluble region to the non-soluble region occurs, and all the heuristics failed to efficiently handle these instances and thus timed out.

4 Conclusion

Labelling heuristics may lead to a substantial reduction of the search space when solving CSP models. However, little is known about the application domains of the known heuristics. This work follows the call of Tsang *et al.* for mapping combinations of algorithms and heuristics to application domains [16]. Rather than inferring the applications domains of (known) algorithm/heuristic combinations,

we here advocate inferring (known or new) algorithm/heuristic combinations for application domains.

Our approach is to first formalise a CSP application domain as a model family, so as to exhibit the generic finite-domain constraint store for all models in that family. By analysing the interaction of an algorithm with this generic constraint store, one can then infer labelling heuristics for that family. Usually, one would at least look for a heuristic that excels at finding the first solution, one that excels at disproving the existence of solutions, and one that detects and handles the phase transition. We here illustrated this approach on a domain of subset problems, as well as on the effect of labelling heuristics for a fixed search algorithm, namely forward checking. We inferred two heuristics for this domain, one for each of the first two kinds.

We generate random instances by iterating over an interval of $\langle n, p_1, k\rangle$ instance classes and generating a suitably sized sample of random instances for each class. For each sample, if the instances are comparable (e.g., all the instances have a solution), the median cost is computed; otherwise (e.g., some instances have a solution but some do not), we cannot judge which heuristic is the "best" for this sample. We then devise a lookup table, where either the "best" heuristic for a given instance class $\langle n, p_1, k\rangle$ is designated [7], or a switching between heuristics is designated because none of the heuristics is considered to be better than another one for this class of instances [11]. This switching can be done by deploying one of the heuristics first, and monitoring the progress so as to switch to the next one in case of thrashing. This lookup table is then used by a meta-heuristic. If this meta-heuristic beats — on the average over numerous instances of the family — the default heuristic of the solver, then this meta-heuristic can become a family-specific and even highly instance-sensitive default heuristic of the solver. If this is repeated for many application domains, then modellers can — if they wish so — be relieved from indicating or implementing a heuristic at modelling-time, which often is a too early commitment anyway, due to the instance-sensitivity of heuristics.

In terms of related work, Figure 3 shows the classical approach to designing heuristics in full lines, whereas the contribution of our approach is emphasised in dashed lines and italicised text. A curved arrow from a full line to a dashed line indicates our replacement of the full line with the dashed line. We thus replace the design of a single heuristic for a CSP model in the presence of a solver (i.e., search algorithm) with the inference of a *set* of heuristics for a model-*family* by *analysis* of the propagation performed by that solver on the family-specific generic constraint store during labelling. Also, in our approach, random instances are generated *only* for the considered *family* (which does not necessarily contain binary CSPs), rather than for arbitrary (binary) CSPs.

Closely related to our work is first Minton's MULTI-TAC system [12], which automatically synthesises an instance-distribution-specific solver, given a high-level model of some CSP and a set of training instances. While MULTI-TAC uses a synthesis-time brute-force approach to generate candidate problem-specific heuristics from a set of heuristics described by a grammar, we propose inferring

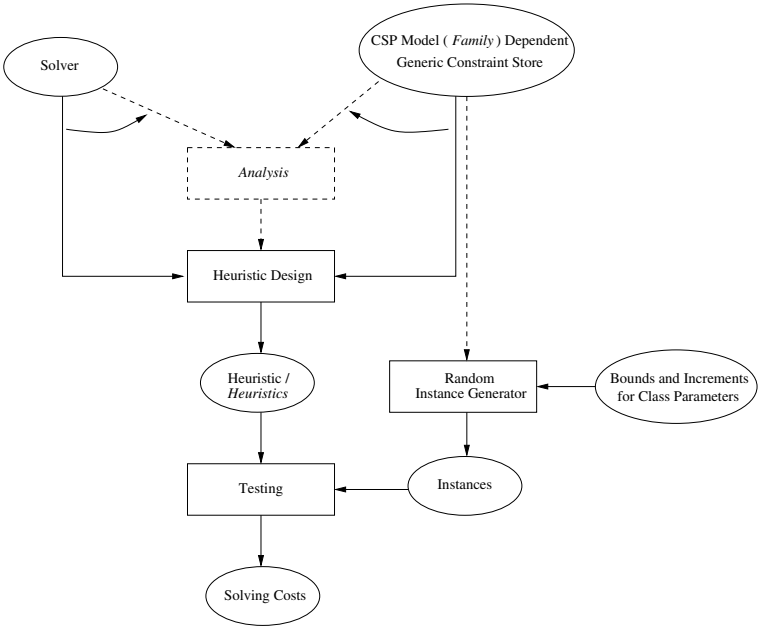


Fig. 3. Contributions to the classical approach to designing heuristics

candidate family-specific heuristics manually by analytically reasoning about the generic constraint store of the family. Second, Sadeh and Fox propose a probabalistic framework for the job shop scheduling domain so as to capture the search space. Based on this framework, a domain specific heuristic is derived [14]. The derived heuristic significantly reduces the search space of the instances used in the experiments. However, the instance sensitivity of heuristics is not tackled, and only one heuristic is derived for the domain.

Our future work includes investigating the superiority or the inferiority of dynamic variable orderings, which choose a variable that is constraining the smallest/largest number of the *future* (yet unassigned) variables, to the here investigated static variable orderings, which choose a variable that is *initially* constraining the smallest/largest number of variables.

We are also planning to investigate other application domains, such as *m-subset problems* (where a maximum of m subsets of a given set have to be found, subject to some constraints), *relation problems* (where a relation between two given sets has to be found, subject to some constraints) [4], *permutation problems* (where a sequence representing a permutation of a given set has to be found, subject to some constraints) [6], and *sequencing problems* (where sequences of bounded size over the elements of a given set have to be found, subject to some constraints) [6], or any combinations thereof.

All results will be built into the compiler of our ESRA constraint modelling language [6,4], which is more expressive than even OPL [17]. This will help us

to fulfill our design objective of also making ESRA more declarative than OPL, without compromising (much) on efficiency.

Acknowledgements. We would like to thank Prof. Edward Tsang (University of Essex, UK) and our colleague Justin Pearson for their invaluable comments. This research is partly funded under grant number 221-99-369 of VR (the Swedish Research Council).

References

1. D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and Y.C. Stamatiou. Random constraint satisfaction: A more accurate picture. In: G. Smolka (ed), *Proc. of CP'97*, pp. 107–120. LNCS 1330. Springer, 1997.
2. P. Codognet and D. Diaz. Compiling constraints in CLP(FD). *J. of Logic Programming* 27(3):185–226, 1996.
3. T. Ellman, J. Keane, A. Banerjee, and G. Armhold. A transformation system for interactive reformulation of design optimization strategies. *Research in Engineering Design* 10(1):30–61, 1998.
4. P. Flener. Towards relational modelling of combinatorial optimisation problems. In: Ch. Bessière (ed), *Proc. of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001.
5. P. Flener and B. Hnich. The syntax and semantics of ESRA. Evolving internal report of the ASTRA Team, at http://www.dis.uu.se/∼pierref/astra/.
6. P. Flener, B. Hnich, and Z. Kızıltan. Compiling high-level type constructors in constraint programming. In: I.V. Ramakrishnan (ed), *Proc. of PADL'01*, pp. 229–244. LNCS 1990. Springer, 2001.
7. P. Flener, B. Hnich, and Z. Kızıltan. A meta-heuristic for subset problems. In: I.V. Ramakrishnan (ed), *Proc. of PADL'01*, pp. 274–287. LNCS 1990. Springer, 2001.
8. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.
9. J.M. Gratch and S.A. Chien. Adaptive problem-solving for large scale scheduling problems: A case study. *J. of Artificial Intelligence Research* 4:365–396, 1996.
10. J.N. Hooker. Testing heuristics: We have it all wrong. *J. of Heuristics* 1:33–42, 1996.
11. Z. Kızıltan and P. Flener. An adaptive meta-heuristic for subset problems. Submitted for review. Available via http://www.dis.uu.se/∼pierref/astra/.
12. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints* 1(1–2):7–43, 1996.
13. T. Müller. Solving set partitioning problems with constraint programming. *Proc. of PAPPACT'98*, pp. 313–332. The Practical Application Company, 1998.
14. N.M. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Arificial Intelligence* 86(1):1–41, 1996.
15. E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
16. E.P.K. Tsang, J.E. Borrett, and A.C.M. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. *Proc. of AISB'95*, pp. 203–216, 1995. IOS Press.
17. P. Van Hentenryck. *The* OPL *Optimization Programming Language*. The MIT Press, 1999.

Addressing the Qualification Problem in FLUX

Yves Martin and Michael Thielscher

Dresden University of Technology

Abstract. The Qualification Problem arises for planning agents in real-world environments, where unexpected circumstances may at any time prevent the successful performance of an action. We present a logic programming method to cope with the Qualification Problem in the action programming language FLUX, which builds on the Fluent Calculus as a solution to the fundamental Frame Problem. Our system allows to plan under the default assumption that actions succeed as they normally do, and to reason about these assumptions in order to recover from unexpected action failures.

1 Introduction

Intelligent agents in open environments inevitably face the Qualification Problem: The executability of an action can never be predicted with absolute certainty; at any time, actions in the real-world may surprisingly fail [13]. Yet it would be irrational, and even impossible in general, for a planning agent to foresee all conceivable reasons for an action to go wrong. Rather, a rational agent needs to devise plans under the assumption that the world will behave as expected. On the other hand, being aware of these assumptions helps an agent to explain and recover from unexpected failures encountered during the execution of a plan.

For a long time, the main theoretical result on the Qualification Problem had been a negative one: While a solution must involve the ability to assume away, by default, so-called abnormal qualifications of actions [14], straightforward minimization of abnormality yields anomalous models [9]. This problem being unsolved, previously developed action programming languages and planning systems, such as [8,16,12,2], did not attempt to address the Qualification Problem. The problem of anomalous models has, however, recently been solved in a formal account of the Qualification Problem presented in [21]. This theory builds on the Fluent Calculus as a predicate logic formalism for reasoning about actions which is one of the standard solutions to the fundamental Frame Problem [18].

In this paper, we integrate the theoretical account of the Qualification Problem into the action programming language FLUX (the *Fluent Calculus Executor*) [20]. Based on constraint logic programming, FLUX allows to specify and reason about actions with incomplete states, and thus to solve planning problems under incomplete information. Its core consists of a logic programming account of the Fluent Calculus solutions to the Frame and Ramification Problem [19,17]. Extending FLUX so as to cope with the Qualification Problem, our system allows

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 290–304, 2001.

the user to specify default assumptions concerning the executability and effects of actions. Plans are then generated under these assumptions, and the system is able to reason about the assumptions made and to withdraw appropriate ones in order to explain and recover from unexpected action failures. The language allows to distinguish between strong qualifications (actions not being executable) and weak ones (actions producing unexpected effects). Furthermore, it supports the specification of preferences among the default assumptions, by which is aided the search for reasonable explanations in case of unexpected action failure.

The paper is organized as follows. In Section 2, we recapitulate the theoretical account of the Qualification Problem in the Fluent Calculus. In Section 3, we show how to extend FLUX to cope with the Qualification Problem. For a more detailed discussion of this section the reader is referred to [11]. In Section 4 an application is described and Section 5 gives a summary.

2 The Qualification Problem in the Fluent Calculus

2.1 Simple State Update Axioms

The simple Fluent Calculus [19] combines, in pure classical logic, the Situation Calculus with a STRIPS-like solution to the representational and inferential Frame Problem. The standard sorts ACTION and SIT (i.e., situations) are inherited from the Situation Calculus [7] along with the standard functions $S_0 : \text{SIT}$ and $Do : \text{ACTION} \times \text{SIT} \mapsto \text{SIT}$ denoting, resp., the initial situation and the successor situation after performing an action; furthermore, the standard predicate $Poss : \text{ACTION} \times \text{SIT}$ denotes whether an action is possible in a situation. To this the Fluent Calculus adds the sort STATE with sub-sort FLUENT along with the pre-defined functions $\emptyset : \text{STATE}$, $\circ : \text{STATE} \times \text{STATE} \mapsto \text{STATE}$, and $State : \text{SIT} \mapsto \text{STATE}$, denoting, resp., the empty state, the union of two states, and the state of the world in a situation. Based on this signature, the Fluent Calculus provides a rigorously logical account of the concept of a state being characterized by the set of fluents that are true in the state. To this end, the following foundational axioms stipulate that function $\circ$ behaves like set union with $\emptyset$ as the empty set:[1]

$$
\begin{array}{rl}
z_1 \circ (z_2 \circ z_3) = (z_1 \circ z_2) \circ z_3 & \neg Holds(f, \emptyset) \\
z_1 \circ z_2 = z_2 \circ z_1 & Holds(f_1, f) \supset f = f_1 \\
z \circ z = z & Holds(f, z_1 \circ z_2) \supset Holds(f, z_1) \vee Holds(f, z_2) \\
z \circ \emptyset = z & (\forall f)\,(Holds(f, z_1) \equiv Holds(f, z_2)) \supset z_1 = z_2 \\
 & (\forall \Phi)(\exists z)(\forall f)\,(Holds(f, z) \equiv \Phi(f))
\end{array}
$$

where Φ is a second-order predicate variable of sort FLUENT and the macro *Holds* means that a fluent is part of a state:

$$Holds(f, z) \stackrel{\text{def}}{=} (\exists z')\, z = f \circ z' \tag{1}$$

[1] Free variables in formulas are assumed universally quantified. Variables of sorts ACTION, SIT, FLUENT, and STATE shall be denoted by the letters a, s, f, and z, resp. The function $\circ$ is written in infix notation.

The very last one of the axioms above stipulates the existence of a state for all possible combinations of fluents. A second macro, which reduces to (1), is used for fluents holding in situations:

$$Holds(f, s) \stackrel{\text{def}}{=} Holds(f, State(s))$$

As an example, consider a blocks world axiomatization using the FLUENT functions $On(x, y)$, $GluedToTable(x)$, and $Has(r, x)$ denoting, resp., whether block x is on y (which could either be another block or the constant $Table$), whether block x is glued to the table, and whether robot r is in possession of x. Suppose, that in the initial state it is known that blocks A and C are on the table and B is on C; that no block y is on top of block A or B; and that robot $Robbie$ is in possession of glue:

$$\begin{array}{l} Holds(On(A, Table), S_0) \wedge Holds(On(C, Table), S_0) \wedge Holds(On(B, C), S_0) \\ \wedge\ (\forall y)\,(\neg Holds(On(y, A), S_0) \wedge \neg Holds(On(y, B), S_0)) \\ \wedge\ Holds(Has(Robbie, Glue), S_0) \end{array} \tag{2}$$

Assuming uniqueness of names for all functions with range FLUENT, the macro definitions and the foundational axioms imply that (2) is equivalent to,

$$\begin{array}{l} (\exists z)\,(\,State(S_0) = On(A, Table) \circ On(C, Table) \\ \qquad\qquad \circ On(B, C) \circ Has(Robbie, Glue) \circ z \\ \quad \wedge\ (\forall y)\,(\neg Holds(On(y, A), z) \wedge \neg Holds(On(y, B), z)) \\ \quad \wedge\ \neg Holds(On(A, Table), z) \wedge \neg Holds(On(C, Table), z) \\ \quad \wedge\ \neg Holds(On(B, C), z) \wedge \neg Holds(Has(Robbie, Glue), z)\,) \end{array} \tag{3}$$

The reader may notice that the constraints on sub-STATE z not only reflect the negated statements in (2) but also the fact that the fluents $On(A, Table)$ etc. do not recur. This will allow to quickly infer the result of removing any of these fluents from $State(S_0)$ as a negative effect.

The Frame Problem is solved in the Fluent Calculus using so-called state update axioms, which specify the difference between the states before and after an action. The axiomatic characterization of negative effects, i.e., facts that become false, is given by this inductive abbreviation, which generalizes STRIPS-style update [3] to incomplete states:

$$z' = z - f \stackrel{\text{def}}{=} [z' \circ f = z \vee z' = z] \wedge \neg Holds(f, z')$$

$$\begin{array}{l} z' = z - (f_1 \circ \ldots \circ f_n \circ f_{n+1}) \stackrel{\text{def}}{=} \\ \quad (\exists z'')\,(z'' = z - (f_1 \circ \ldots \circ f_n) \wedge z' = z'' - f_{n+1}) \end{array}$$

On this basis, the following is the general form of a state update axiom for a (possibly nondeterministic) action $A(\boldsymbol{x})$ with a bounded number of (possibly conditional) effects:

$$\begin{aligned} Poss(A(\boldsymbol{x}), s) \supset\ & (\exists \boldsymbol{y}_1)\,(\Delta_1 \wedge State(Do(A(\boldsymbol{x}), s)) = (State(s) \circ \vartheta_1^+) - \vartheta_1^-) \\ & \vee \ldots \vee \\ & (\exists \boldsymbol{y}_n)\,(\Delta_n \wedge State(Do(A(\boldsymbol{x}), s)) = (State(s) \circ \vartheta_n^+) - \vartheta_n^-) \end{aligned} \tag{4}$$

where the sub-formulas $\Delta_i(\boldsymbol{x}, \boldsymbol{y}_i, State(s))$ specify the conditions on $State(s)$ under which $A(\boldsymbol{x})$ has the positive and negative effects ϑ_i^+ and ϑ_i^-, resp. Both ϑ_i^+ and ϑ_i^- are STATE terms composed of fluents with variables among $\boldsymbol{x}, \boldsymbol{y}_i$.[2]

Consider, e.g., the ACTION terms $Move(r, u, v, w)$ and $GlueToTable(r, x)$, denoting the action of robot r moving block u away from v onto w, and gluing block x to the table, resp. The direct effects of these two actions can be defined by these state update axioms:

$$\begin{aligned} & Poss(Move(r, u, v, w), s) \supset \\ & \quad State(Do(Move(r, u, v, w), s)) = (State(s) \circ On(u, w)) - On(u, v) \\ & Poss(GlueToTable(r, x), s) \supset \\ & \quad State(Do(GlueToTable(r, x), s)) = State(s) \circ GluedToTable(x) \end{aligned} \tag{5}$$

Put in words, after moving u it is on w and no longer on v, and after gluing x this block is glued to the table. Recall specification (2), and suppose, for the sake of argument, that $Poss(Move(Robbie, A, Table, B), S_0)$. Let $S_1 = Do(Move(Robbie, A, Table, B), S_0)$. Then the state update axiom for $Move$ in (5) implies

$$State(S_1) = (State(S_0) \circ On(A, B)) - On(A, Table)$$

Replacing $State(S_0)$ by an equal term according to (3) yields, after applying the macro for negative effects and performing simplification,

$$(\exists z)\, State(S_1) = On(C, Table) \circ On(B, C) \circ Has(Robbie, Glue) \circ z \circ On(A, B)$$

We have now obtained from an incomplete initial specification a still partial description of the successor state, which in particular includes the unaffected fluents $On(C, Table)$, $On(B, C)$, and $Has(Robbie, Glue)$. These fluents have thus survived the computation of the effect of the action and so need not be carried over by separate axioms now. Moreover, knowledge specified in (3) as to which fluents do not hold in z applies to the new state, which includes z, just as well. Thus, all unchanged fluent values have been concluded to persist without applying extra inference steps.

2.2 State Update Axioms with Ramifications

In the Fluent Calculus for ramifications, indirect effects are inferred by the successive application of so-called causal relationships, which state under what conditions an effect triggers another one [17]. A causal relationship is formally specified with the help of the expression $Causes(\varepsilon, \varrho, z, s)$ where ε (the *triggering*

[2] If the conditions Δ_i are not mutually exclusive, then the action is nondeterministic.

effect) and ϱ (the *ramification*, i.e., indirect effect) are possibly negated atomic fluent formulas and z is a state and s a situation. The intuitive meaning is that the change to ε *causes* the change to ϱ in state z and situation s.

For example, let $\mathsf{Ab}(\mathit{Movable}(x), \mathit{Glued})$ be a new FLUENT, denoting an abnormality wrt. block x being movable due to the fact that x is glued to the table.[3] The following *state constraint* relates this fluent in the obvious way to the fluent $\mathit{GluedToTable}(x)$:

$$\mathit{Holds}(\mathsf{Ab}(\mathit{Movable}(x), \mathit{Glued}), s) \equiv \mathit{Holds}(\mathit{GluedToTable}(x), s) \tag{6}$$

Two accompanying causal relationships specify the causal dependence that a block x will become immovable if it gets glued to the table, and that this abnormal qualification will disappear if the block gets freed somehow:

$$\begin{array}{l} \mathit{Causes}(\mathit{GluedToTable}(x), \mathsf{Ab}(\mathit{Movable}(x), \mathit{Glued}), z, s) \\ \mathit{Causes}(\neg\mathit{GluedToTable}(x), \neg\mathsf{Ab}(\mathit{Movable}(x), \mathit{Glued}), z, s) \end{array} \tag{7}$$

On the basis of causal relationships, the Ramification Problem is solved by causally propagating indirect effects: Starting from the direct effects of an action, causal relationships are applied successively. The overall result of performing the action is then a fixpoint of such a chain of indirect effects. Formally, in state update axioms for ramifications the simple equations $\mathit{State}(\mathit{Do}(A(\boldsymbol{x}), s)) = (\mathit{State}(s) \circ \vartheta_i^+) - \vartheta_i^-$ as in (4) are replaced by sub-formulas of this form:

$$z = (\mathit{State}(s) \circ \vartheta_i^+) - \vartheta_i^- \supset \mathit{Ramify}(z, \vartheta_i^+, \vartheta_i^-, \mathit{Do}(A(\boldsymbol{x}), s))$$

where $\mathit{Ramify}(z, e^+, e^-, s)$ means that $\mathit{State}(s)$ is a fixpoint of iteratively applying causal relationships to state z and effects e^+, e^- in situation s. (We refer to [21] for the formal definition of *Ramify* by a second-order axiom.)

2.3 Qualifications in the Fluent Calculus

The theoretical account of the Qualification Problem uses the binary function $\mathsf{Ab}(x, y)$ whose range is the sort FLUENT. The first argument, x, denotes properties like $\mathit{Movable}(u)$ or $\mathit{Functioning}(\mathit{Gripper\text{-}of}(r))$. The second argument, y, indicates the cause for the abnormality. For convenience, we use the macros $\mathsf{Ab}(x, z)$ and $\mathsf{Ab}(x, s)$ to represent that for some y, $\mathsf{Ab}(x, y)$ holds in state z and situation s, respectively:

$$\mathsf{Ab}(x, z) \stackrel{\text{def}}{=} (\exists y)\, \mathit{Holds}(\mathsf{Ab}(x, y), z) \qquad \mathsf{Ab}(x, s) \stackrel{\text{def}}{=} \mathsf{Ab}(x, \mathit{State}(s))$$

Instances of the generic 'abnormality' fluent are used to summarize the abnormal qualifications of actions, that is, obstacles which are *a priori* unlikely to happen and therefore need to be assumed away by default in order to jump to the conclusion that the action is possible under normal circumstances. E.g., in the

[3] The special fluent Ab will play a key role in our account for the Qualification Problem later in this paper.

light of abnormal qualifications, the preconditions for our example actions are specified by,

$$\begin{array}{l} Poss(Move(r,u,v,w),s) \equiv \\ \quad u \mathrel{/\!\!=} w \wedge v \mathrel{/\!\!=} w \wedge Holds(On(u,v),s) \\ \quad \wedge\ (\forall y)\,(\neg Holds(On(y,u),s) \wedge \neg Holds(On(y,w),s)) \\ \quad \wedge \neg \mathsf{Ab}(Movable(u),s) \wedge \neg \mathsf{Ab}(Functioning(Gripper\text{-}of(r)),s) \end{array} \tag{8}$$

$$\begin{array}{l} Poss(GlueToTable(r,x),s) \equiv \\ \quad Holds(Has(r,Glue),s) \wedge Holds(On(x,Table),s) \wedge (\forall y)\,\neg Holds(On(y,x),s) \\ \quad \wedge \neg \mathsf{Ab}(Movable(x),s) \wedge \neg \mathsf{Ab}(Functioning(Gripper\text{-}of(r)),s) \end{array} \tag{9}$$

As illustrated in the previous section, abnormal qualifications that have been caused by the agent himself are accounted for by suitable causal relationships, which is how the general problem of anomalous models is overcome; see [21]. To account for abnormal qualifications other than those caused by the agent, instances of Ab are allowed to become true during any situation transition as a side effect of the mere fact that the very transition takes place. So doing requires additional causal relationships, which, as opposed to those shown in (7), describe exogenously caused abnormalities. These are modeled using the predicates $ExogCaused(f,s)$ and $ExogUncaused(f,s)$, indicating that in situation s fluent f arises (resp. vanishes) due to an exogenous cause. The effect of exogenous causes is specified by corresponding causal relationships.

Up to this point the treatment of qualifications did not affect the monotonicity of the solution to the Frame and Ramification Problem. A nonmonotonic component, however, is required to *minimize* abnormal qualifications whenever they are not caused by an action that has been performed. This is achieved by adding appropriate default rules in the sense of [15], by which the Fluent Calculus gets embedded into a default theory. Formally, exogenous influence on abnormalities is minimized by default rules of the following form:

$$\frac{:\ \neg ExogCaused(\mathsf{Ab}(x,Exog),s)}{\neg ExogCaused(\mathsf{Ab}(x,Exog),s)} \qquad \frac{:\ \neg ExogUncaused(\mathsf{Ab}(x,Exog),s)}{\neg ExogUncaused(\mathsf{Ab}(x,Exog),s)} \tag{10}$$

An accompanying default assumption concerns abnormalities of any kind in the *initial* situation. Their minimization is carried out by defaults of the following form:

$$\frac{:\ \neg Holds(\mathsf{Ab}(x,y),S_0)}{\neg Holds(\mathsf{Ab}(x,y),S_0)} \tag{11}$$

If, e.g., the observations suggest no abnormalities initially, then the underlying default theory has a unique extension (in the sense of [15]), which includes $(\forall x)\,\neg\mathsf{Ab}(x,S_0)$. (Recall that $\neg\mathsf{Ab}(x,s)$ means $\neg Holds(\mathsf{Ab}(x,y),s)$ for any y.) Hence, (8) implies that $Move(Robbie,A,Table,B)$ is possible in S_0 given initial state (2). If this action nonetheless fails, that is, if the observation $\neg Poss(Move(Robbie,A,Table,B),S_0)$ is added, then the default theory admits different extensions. These are obtained by applying all defaults except for one

instance of (11) with either $\{x/Movable(A)\}$ or $\{x/Functioning(Gripper\text{-}of(Robbie))\}$. As will be shown in Section 3.5 below, this approach can also be used to account for so-called weak qualifications, that is, unexpected effects of actions. Furthermore, by appealing to *prioritized* default logic [1], one can specify qualitative knowledge of the relative likelihood of the various explanations for abnormal qualifications (cf. Section 3.4 below). The accompanying concept of *preferred* extensions helps selecting the most reasonable explanations in case of unexpected action.

3 Addressing the Qualification Problem in FLUX

3.1 Basic FLUX

Our system is implemented in the Eclipse-Prolog system with constraints. It is an extension of the programming language Fluent Calculus Executor (FLUX), a recent implementation of the Fluent Calculus based on Constraint Logic Programming [20]. The distinguishing feature of FLUX is to support incomplete states, which are modeled by open lists of the form

```
Z0 = [F1,...,Fm | Z]
```

(encoding the state description $\texttt{Z0} = \texttt{F1} \circ \ldots \circ \texttt{Fm} \circ \texttt{Z}$), along with constraints

```
not_holds(F, Z)
not_holds_all([X1,...,Xk], F, Z)
```

encoding, resp., the negative statements $(\exists \boldsymbol{y})\neg Holds(\texttt{F},\texttt{Z})$ (where $\boldsymbol{y}$ are the variables occurring in F) and $(\exists \boldsymbol{y})(\forall \texttt{X1},\ldots,\texttt{Xk})\neg Holds(\texttt{F},\texttt{Z})$ (where $\boldsymbol{y}$ are the variables occurring in F except $\texttt{X1},\ldots,\texttt{Xk}$). These two constraints are used to bypass the problem of "negation as failure" for incomplete states. In order to process these constraints, so-called declarative Constraint Handling Rules [4] have been defined and proved correct under the foundational axioms of the Fluent Calculus. In addition, the core of FLUX contains definitions for `holds(F,Z)`, by which is encoded macro (1), and `update(Z1,ThetaP,ThetaN,Z2)`, which encodes the state equation $\texttt{Z2} = (\texttt{Z1} \circ \texttt{ThetaP}) - \texttt{ThetaN}$. The following is an encoding in FLUX of the precondition (8) and state update axiom (5) with ramifications of the action *Move*,

```
poss(move(R, U, V, W), Z) :-
   is_robot(R), is_block(U), is_block(V), is_block(W), U\=W, V\=W,
   not_holds_all(Y, on(Y, U), Z), holds(on(U, V), Z),
   not_holds_all(Y, on(Y, W), Z),
   not_holds_all(Y, ab(mov(U), Y), Z),
   not_holds_all(Y, ab(func(grip(R)), Y), Z).

state_update(Z1, move(R,U,V,W), Z2, S, H) :-
   update(Z1, [on(U,W)], [on(U,V)], Z3),
   ramify(Z3, [on(U,W)], [on(U,V)], Z2, S, H)).
```

where the variable H stands for the history list as defined below.

3.2 Overview

Our system implements the defaults of the underlying default theory without the need of any special theorem prover. Rather, a modified version of the planning algorithm together with the internal Eclipse-Prolog inference mechanisms is used to construct the extensions of the underlying default theory, which entail the possible explanations for unexpected action failures. This also means that if a plan can be executed without any exceptions to the normal execution of actions then no additional computations are needed. In this case, the implementation will behave like any other system without an approach to the Qualification Problem.

In this work, we have used a search process with an associated level in order to find the most likely explanations first. In terms of the underlying default theory (with priorities among defaults) this means to search for the least preferred default that does not apply. The system also replaces previously considered explanations in case a default is no longer preferred in context of a formerly established explanation. Of course, the replacement is only performed if it is consistent with the executed action sequence.

In the following we sketch the course of the program and give references to the next subsections:

1. **Definition of the Task**
   - 1.1 **Define the initial state.** This operation includes the verification that the state is consistent wrt. the state constraints.
   - 1.2 **Define the goal state.**
2. **The Planning Algorithm**
   - 2.1 **Find a plan.** State updates are computed with ramifications (cf. Section 2.2). An agent has no influence on exogenously caused abnormalities and cannot yet have caused any abnormal qualification in the initial situation. Therefore, all defaults on the absence of exogenous causes are assumed to apply during the planning process. Iterative deepening is applied as search strategy. Furthermore, the program uses some heuristics to cut down the search space.
   - 2.2 **Double-check the computed plan.** Planning with incomplete state information requires to verify an established plan against both, not achieving the goal and not being executable.
3. **Plan Execution**
   - 3.1 **Execute the computed plan step by step.** The execution of each action is monitored. If no action fails and all the actions achieve their intended effects then the goal state will be reached and the program terminates. Otherwise, the program proceeds with step 4.
4. **Explanation of and Recovery from Action Failures**
   - 4.1 **Search for an explanation for the unexpected observation.** If an action surprisingly fails or does not produce all the intended effects then an abnormal qualification must have occurred. This means that at least one default of the underlying default theory can no longer be applied. For the intended applications of the program it is in most cases sufficient to search for atomic explanations, i.e., where the application

of exactly one default is blocked during the construction of each of the possible extensions. Furthermore, we assume that there is always at most one explanation at each level. Using the built-in inference mechanisms of the Eclipse-Prolog system, each extension of the underlying default theory is considered where one instance of a default rule is blocked. If the non-application of such a default rule entails the observation then an explanation has been found and the search process stops (cf. Section 3.3).

4.2 **Find the explanation with the highest priority first.** Using a search process with levels, the search will find only explanations with a priority higher or equal to the current level of the search algorithm (cf. Section 3.4). Only if there is no such very likely explanation that accounts for the observation then the program searches with the next lower level and thus considers less likely explanations.

4.3 **Determine the current state and replan.** The search process of the program can deliver a strong or a weak qualification (cf. Section 3.5) as an explanation. In both cases, this new information is integrated into the current state, which becomes the initial state of the new planning problem. Then the planning algorithm is used to find a new plan despite the encountered abnormality. Hence, the program proceeds with step 2.

An important concept in the approach to the Qualification Problem in FLUX is the notion of a *history list*. Such a list contains all the abnormalities that have occurred so far during the execution of the program. Each entry in the list has three parts. The first part describes the abnormality predicate with the property and the cause. In the second part the situation, in which the abnormality occurred, is stated. The third part denotes the possible observation which lead to the occurrence of the abnormality. In addition to these entries, the history list contains the current level of the search process. The history list is modeled by an open list with a tail variable.

3.3 Constructions of Extensions

In this section we show how extensions of the underlying default theory are constructed for strong qualifications of actions. The inference process is similar for weak qualifications.

Effects with an exogenous cause are implemented by causal relationships of the following form:

```
causes(_, ab(mov(X), exog), Z, S, H) :- block(X),
 X\= table, exogcaused(ab(mov(X), exog), S, H).

causes(_, -(ab(mov(X), exog)), Z, S, H) :- block(X),
 X\=table, exoguncaused(-(ab(mov(X), exog)), S, H).
```

Please note that the indirect effects $\mathsf{Ab}(\mathit{Movable}(x), \mathit{Exog})$ in these clauses are not conditioned on any direct effect. Consequently, these positive or negative indirect effects occur as a side effect of every transition whenever the predicates $\mathit{ExogCaused}(\mathsf{Ab}(x, \mathit{Exog}), s, h)$ or $\mathit{ExogUncaused}(\mathsf{Ab}(x, \mathit{Exog}), s, h)$ hold, where

the variables s and h stand for the considered situation and history list, respectively. Indirect effects wrt. the abnormality $\mathsf{Ab}(\mathit{Functioning}(\mathit{Gripper\text{-}of}(r)), \mathit{Exog})$ are similarly encoded.

Predicate *ExogCaused* and predicate *ExogUncaused* are specified in a similar way in our program. For brevity we present only the important details of the definition for *ExogCaused* .

```
exogcaused(AB, S, H) :-  ...
                         top(H, H2),
                         length1(H2,0),
                         holds(h(AB, S), H)))).
```

The clause uses the auxiliary predicates $\mathit{Top}(h_1, h_2)$ and $\mathit{Length1}(h, n)$. The predicate *Top* takes the present history list and yields the currently considered abnormality predicate, if any. The predicate *Length1* delivers the length of a list. The definition of the clause ensures that exactly one abnormality is considered as an explanation at any stage of the search process.

Extensions are constructed during state updates with ramifications. For each action all possible extensions of the underlying default theory are tried until the established extension accounts for the observation. This is achieved by blocking the application of each default one after the other. The *ExogCaused* or the *ExogUncaused* predicate is assumed to hold and the specific default is blocked by means of adding the corresponding abnormality as indirect effect to the current state. The addition is performed by the clauses for causal relationships together with the clauses for state update axioms with ramifications. The predicates *ExogCaused* and *ExogUncaused* ensure that only one default is blocked at a time, and the Eclipse-Prolog SLDNF-resolution with backtracking yields all the extensions. The possible defaults for the initial situation are computed in the same way. To this end, the special constant " ϵ " (read "no-op") is introduced. It denotes the empty action without any positive or negative direct effects. This action is always possible and is only executed as the very first action.

As illustration, consider the initial state as specified in (2) together with the following definitions for blocks and robots:

```
is_robot(robbie).            is_block(table). is_block(a).
                             is_block(b).     is_block(c).
```

The query $\mathit{Init}(z_0, h), \mathit{Res}(\epsilon, z_0, S_0, z_1, s_1, h)$, $\backslash{+}\mathit{Poss}(\mathit{Move}(\mathit{Robbie}, A, \mathit{Table}, B), z_1)$ admits two computed answer substitutions, where the predicate *Init* denotes the initial state and the predicate *Res* denotes the execution of exactly one action:

$$\{z_1/[\mathsf{Ab}(\mathit{Movable}(A), \mathit{Exog}), \mathit{On}(A, \mathit{Table}), \mathit{On}(B, C), \mathit{On}(C, \mathit{Table}), \mathit{Has}(\mathit{Robbie}, \mathit{Glue}) \mid z],\ h/[H(\mathsf{Ab}(\mathit{Movable}(A), \mathit{Exog}), S_0) \mid h_1]\}$$

$$\{z_1/[\mathsf{Ab}(\mathit{Functioning}(\mathit{Gripper\text{-}of}(\mathit{Robbie})), \mathit{Exog}), \mathit{On}(A, \mathit{Table}), \mathit{On}(B, C), \mathit{On}(C, \mathit{Table}), \mathit{Has}(\mathit{Robbie}, \mathit{Glue}) \mid z],\ h/[H(\mathsf{Ab}(\mathit{Functioning}(\mathit{Gripper\text{-}of}(\mathit{Robbie})), \mathit{Exog}), S_0) \mid h_1]\}$$

These substitutions are computed by applying all defaults except for one instance of (11) with $\{x/Movable(A)\}$ for the first substitution and $\{x/Functioning(Gripper\text{-}of(Robbie))\}$ for the second one. This way, our program determines the two extensions in this example as described at the end of Section 2.3.

Extensions are constructed using the inference mechanism of the Eclipse-Prolog system, i.e., an implementation of the SLDNF-resolution. This resolution scheme is sound. For the soundness proof it should be referred to Lloyd [10].

In our implementation extensions are constructed using the 9-ary predicate *RunDefault* . We present a schematic version of its encoding:

```
rundefault([], Z, Z, S, S, SW, H1, H2, R) :-
 \+ poss(R, Z), ...
rundefault([F|L], Z0, Z, S, SF, SW, H1, H2, R) :-
 append([A], [E], F), !, ...
 res(A, Z0, S, Z1, S1, H1), ...
 current(S1, H1, HH), (HH=[]; HH=[h(_,_,o(M,_))], \+ poss(M, Z1)),
 ...
 rundefault(L, Z2, Z, S1, SF, SW, H1, H2, R).
```

The predicate *RunDefault* has a recursive definition. The recursion is performed on its first argument. This argument represents the executed sequence of actions as a list. The last argument of *RunDefault* stores the current observation. It is the action that failed unexpectedly for the reason of a strong abnormal qualification. Thus, for the predicate to terminate successfully the computed explanation must account for the observation after having performed the complete sequence of actions. Of course, all other observations recorded in the history list must also be taken into account during the search. To this end, the auxiliary predicate $Current(s, h_1, h_2)$ is used. It checks for the current situation of the search process, whether an unexpected action failure has occurred in this situation during the execution of the plan. If this is the case then the predicate *Current* delivers the corresponding action. Otherwise, the empty list is returned.

3.4 Selection of the Preferred Extension

Extensions are constructed in accordance with the underlying set-prioritized default logic, which is an extension of the prioritized default logic and can be used to define preferences between defaults [1,21]. The preference relations in the program are defined with or without context-dependency. The first case defines a set-preference ordering among defaults so that any two instances of a default concerning the same object are compared to an instance of another default concerning the object. If there is no context of a previous explanation then the second case gives a definition of general preference between defaults. As illustration, consider the implementation for the running example together with the following preference relations:

```
level([12,11,10,9]).

preference(ab(mov(_), _), nc,  9).
```

```
preference(ab(func(grip(_)), _), nc,  10).
preference(ab(mov(_), _), ab(func(grip(_)), _), 11).
```

That is, without context the explanation $\mathsf{Ab}(\mathit{Functioning}(\mathit{Gripper\text{-}of}(x)), y)$ is more likely than the explanation $\mathsf{Ab}(\mathit{Movable}(x), y)$. In contrast, the explanation $\mathsf{Ab}(\mathit{Movable}(x), y)$ is preferred over the assumption of two explanations of the form $\mathsf{Ab}(\mathit{Functioning}(\mathit{Gripper\text{-}of}(x)), y)$.

The preference relations are taken into account when using the predicates *ExogCaused* and *ExogUncaused*. The following shows the part of the clause without context information for the predicate *ExogCaused*:

```
exogcaused(AB, S, H) :-  ...
                         preference(AB, nc, P),
                         member(l(D), H), !, P>=D,
                         top(H, H2), ...
```

The priority of the currently considered explanation is obtained. Afterwards, the current level of the search is obtained from the history list. Further on, the computed priority of the considered explanation is compared to this level. If the priority is at least as high as the current level then the procedure continues as described in Section 3.3.

The change to the next lower level in the search is encoded as:

```
..., level(LEVEL), member(LE, LEVEL), changed(LE, H, H3),
rundefault(L2, ZN, ZF, s0, SV, SW, H3, HF, W), ...
```

The auxiliary predicate *ChangeD* sets the current level in the history list. If the predicate *RunDefault* fails then the Eclipse-Prolog system backtracks and the predicate *Member* chooses the next lower level for the search process.

For example, consider the query $\mathit{Init}(z_0, h)$, $\mathit{Level}(\mathit{level})$, $\mathit{Member}(\mathit{le}, \mathit{level})$, $\mathit{ChangeD}(\mathit{le}, h, h_1)$, $\mathit{Res}(\epsilon, z_0, S_0, z_1, s_1, h_1)$, $\backslash{+}\mathit{Poss}(\mathit{Move}(\mathit{Robbie}, A, \mathit{Table}, B), z_1)$, where the initial state is specified as in (2) and the preferences are defined as above. All preferred extensions of the underlying default theory entail $\mathsf{Ab}(\mathit{Functioning}(\mathit{Gripper\text{-}of}(\mathit{Robbie})), S_0)$. In accordance with the preferred extension the query yields the computed answer substitution:

$$\begin{aligned}\{z_1/[&\mathsf{Ab}(\mathit{Functioning}(\mathit{Gripper\text{-}of}(\mathit{Robbie})), \mathit{Exog}), \mathit{On}(A, \mathit{Table}),\\ &\mathit{On}(B, C), \mathit{On}(C, \mathit{Table}), \mathit{Has}(\mathit{Robbie}, \mathit{Glue}) \mid z],\\ h/[&\mathit{le}(10), H(\mathsf{Ab}(\mathit{Functioning}(\mathit{Gripper\text{-}of}(\mathit{Robbie})), \mathit{Exog}), S_0) \mid h_1]\}\end{aligned}$$

3.5 Weak Qualifications

Weak qualifications, that is, failure to produce expected effects, are denoted and minimized in the same way as strong qualifications. Causal relationships and preference relations for weak qualifications are implemented as illustrated by the following clauses:

```
causes(_, ab(trans(X), exog), Z, S, H) :- block(X),
 X\= table, exogcaused(ab(trans(X), exog), S, H).

preference(ab(trans(_), _), nc, 11).
```

The weak qualification $\mathsf{Ab}(\mathit{Transportable}(x), y)$ means that block x is slippery and will slip out of the gripper when transported over long distances. The construction of extension for default theories, which contain defaults with abnormality predicates regarding weak qualifications of actions, is performed in a similar fashion as described in Section 3.3. The information given in Section 3.4 regarding the preferred extension also holds for weak qualifications.

Fluents denoting weak qualifications strengthen the antecedents of state update axioms. This is in contrast to strong qualifications where abnormality predicates occur in action precondition axioms. Thus, a suitable state update axiom with a possible weak qualification for the action $\mathit{Move}(r, u, v, w)$ is encoded as:

```
state_update(Z1, move(R,U,V,W), Z2, S, H) :-
 (not_holds_all(Y, ab(trans(U), Y), Z1), !,
   update(Z1, [on(U,W)], [on(U,V)], Z3),
   ramify(Z3, [on(U,W)], [on(U,V)], Z2, S, H));
 (holds(ab(trans(U), Y), Z1),
  (V\=table,
     update(Z1, [on(U,table)], [on(U,V)], Z3),
     ramify(Z3, [on(U,table)], [on(U,V)], Z2, S, H);
   V==table, ramify(Z1, [], [], Z2, S, H))).
```

For this state update axiom let us consider the query $\mathit{Init}(z_0, h), \mathit{Res}(\epsilon, z_0, S_0, z_1, s_1, h), \ \mathit{Res}(\mathit{Move}(\mathit{Robbie}, B, C, A), z_1, s_1, z_2, s_2, h), \mathit{NotHolds}(\mathit{On}(B, A), z_2)$, where the initial state is specified as in (2). This query yields the computed answer substitution:

$$\{z_2/[\mathit{On}(B, \mathit{Table}), \mathsf{Ab}(\mathit{Transportable}(B), \mathit{Exog}), \mathit{On}(A, \mathit{Table}), \mathit{On}(C, \mathit{Table}), \mathit{Has}(\mathit{Robbie}, \mathit{Glue}) \,|z], \ h/[H(\mathsf{Ab}(\mathit{Transportable}(B), \mathit{Exog}), S_0) \,|\, h_1]\}$$

Thus, the program concluded that the weak qualification $\mathsf{Ab}(\mathit{Transportable}(B), \mathit{Exog})$ occurred, and that block B can be found somewhere on the table.

4 Experiments

Our program has been tested with a LEGO® MINDSTORM™ robot in a delivery scenario. The main component of such a robot is a programmable brick. It is referred to as RCX (Robotic Command Explorer) and has as its core a Hitachi H8 microcontroller. Our robot has a light sensor and a pushbutton sensor and two motors attached to the input ports and output ports of the RCX, respectively. An infrared port is used for the communication between the computer and the RCX while a user program is running on the RCX. Such programs only realize simple behaviours like following a line.

All high level control is performed by the Eclipse-Prolog system. This includes the planning process and the monitoring of the executions of actions by means of exogenous events. In case of an unexpected action failure, the system searches for explanations. The robot only executes primitive actions. Additionally, it observes the occurrence of exogenous events and reports them to the Eclipse-Prolog system. The communication between the system and the robot is achieved through message exchange using a module from the Legolog system [6].

The robot is supposed to deliver objects from one office to another, where a cardboard with bright markers as offices and black lines as tracks denotes the floor plan. The robot has solved the task in our example scenario if there are no more requests in the current state and the robot has returned to its initial position. Two kinds of abnormalities with an appropriate preference relations between them have been defined for this scenario.

For this example domain our program was able to find explanations for unexpected action failures and to recover from them. In the scenario the system concluded that the robot had missed a marker long before the observance of an exception. In the search process all previously executed actions and related observations were taken into consideration to generate the most likely explanation first. With the established explanations the program was able to infer the robot's current position and to find a new plan to solve the task.

5 Summary

We have presented an extension of the action programming language FLUX which copes with the Qualification Problem. Our approach builds on the theoretical work of [21], where the Fluent Calculus has been embedded into a default theory to account for abnormal qualifications of actions. Our system allows to generate plans under the assumption that actions succeed as they normally do, and to reason about these assumptions in order to recover from unexpected action failures. Furthermore, it supports the specification of preferences among the default assumptions, by which is aided the search for reasonable explanations in case of unexpected action failure. While action programming languages have been extended by execution monitoring in the past, e.g., [5], our system is the first which is based on a formal approach to the Qualification Problem. It thus provides a declarative approach to troubleshooting. The crucial advantage of our approach is that explaining unexpected action failures is carried out on the basis of the same action specifications and reasoning techniques which are used for planning.

References

1. Gerhard Brewka. Adding priorities and specificity to default logic. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *Proc. of the European Workshop on Logics in AI (JELIA)*, volume 838 of *LNAI*, pages 247–260, York, UK, September 1994. Springer.
2. Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. Temporal action logics (TAL): Language specification and tutorial. *Electronic Transactions on Artificial Intelligence*, 2(3–4):273–306, 1998. `http://www.ep.liu.se/ea/cis/1998/015/`.

3. Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
4. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
5. Giuseppe De Giacomo, Ray Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot programs. In Cohn, Schubert, and Shapiro, editors, *Proc. of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 453–464, Trento, Italy, June 1998.
6. Hector Levesque and Maurice Pagnucco. Legolog: Inexpensive experiments in cognitive robotics. In *Cognitive Robotics Workshop at ECAI*, pages 104–109, Berlin, Germany, August 2000.
7. Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for a calculus of situations. *Electronic Transactions on Artificial Intelligence*, 3(1–2):159–178, 1998. `http://www.ep.liu.se/ea/cis/1998/018/`.
8. Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997.
9. Vladimir Lifschitz. Formal theories of action (preliminary report). In J. McDermott, editor, *Proc. of IJCAI*, pages 966–972, Milan, Italy, August 1987. Morgan Kaufmann.
10. John W. Lloyd. *Foundations of Logic Programming*. Series Symbolic Computation. Springer, second, extended edition, 1987.
11. Yves Martin. Solving the Qualification Problem in FLUX. Master's thesis, TU Dresden, Germany, March 2001. `http://www.cl.inf.tu-dresden.de/~yves`.
12. Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Proc. of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 212–223, Trento, Italy, June 1998. Morgan Kaufmann.
13. John McCarthy. Epistemological problems of artificial intelligence. In *Proc. of IJCAI*, pages 1038–1044, Cambridge, MA, 1977. MIT Press.
14. John McCarthy. Applications of circumscription to formalizing common-sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.
15. Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.
16. Murray Shanahan. Event calculus planning revisited. In *Proc. of the European Conference on Planning (ECP)*, volume 1348 of *LNAI*, pages 390–402. Springer, 1997.
17. Michael Thielscher. Ramification and causality. *Artificial Intelligence*, 89(1–2):317–364, 1997.
18. Michael Thielscher. Introduction to the Fluent Calculus. *Electronic Transactions on Artificial Intelligence*, 2(3–4):179–192, 1998. `http://www.ep.liu.se/ea/cis/1998/014/`.
19. Michael Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299, 1999.
20. Michael Thielscher. The fluent calculus: A specification language for robots with sensors in nondeterministic, concurrent, and ramifying environments. Technical Report CL-2000-01, Artificial Intelligence Institute, Department of Computer Science, Dresden University of Technology, 2000.
21. Michael Thielscher. The qualification problem: A solution to the problem of anomalous models. *Artificial Intelligence*, 2001.

Extracting Situation Facts from Activation Value Histories in Behavior-Based Robots*

Frank Schönherr, Mihaela Cistelecan**, Joachim Hertzberg, and Thomas Christaller

GMD – German National Research Center for Information Technology
Institute for Autonomous intelligent Systems (AiS)
Schloss Birlinghoven, 53754 Sankt Augustin, Germany

Abstract. The paper presents a new technique for extracting symbolic ground facts out of the sensor data stream in autonomous robots for use under hybrid control architectures, which comprise a behavior-based and a deliberative part. The sensor data are used in the form of time series curves of behavior activation values. Recurring patterns in individual behavior activation curves are aggregated to well-defined patterns, like edges and levels, called *qualitative activations.* Sets of qualitative activations for different behaviors occurring in the same interval of time are summed to *activation gestalts.* Sequences of activation gestalts are used for defining *chronicles,* the recognition of which establishes evidence for the validity of *ground facts.* The approach in general is described, and examples for a particular behavior-based robot control framework in simulation are presented and discussed.

1 Background and Overview

There are several good reasons to include a behavior-based component in the control of an autonomous mobile robot. There are equally good reasons to include in addition a deliberative component. Having components of both types results in a *hybrid* control architecture, intertwining the behavior-based and the deliberative processes that go on in parallel. Together, they allow the robot to react to the dynamics and unpredictability of its environment without forgetting the high-level goals to accomplish. Arkin [Ark98, Ch. 6] presents a detailed argument and surveys hybrid control architectures; the many working autonomous robots that use hybrid architectures include the Remote Agent Project [MNPW98,Rem00] as their highest-flying example.

While hybrid, layered control architectures for autonomous robots, such as Saphira [KMSR97] or 3T [BFG+97] are state of the art, some problems remain that make it a still complicated task to build a control system for a concrete robot to work on a concrete task. To quote Arkin [Ark98, p. 207],

> the nature of the boundary between deliberation and reactive execution is not well understood at this time, leading to somewhat arbitrary architectural decisions.

* This work is partially supported by the German Fed. Ministry for Education and Research (BMBF) in the joint project AgenTec (521-75091-VFG 0005B).

** On leave from Tech. University of Cluj-Napoca, Romania. The work was supported by a Roman-Herzog scholarship of the Alexander von Humboldt foundation

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 305–319, 2001.

One of the problems is to keep up-to-date the symbolic world representation for the deliberative component. There are solutions to important parts of that problem, such as methods and algorithms for sensor-based localization to reason about future navigation actions: [FBT99] presents one of the many examples for on-line robot pose determination based on laser scans. If the purpose of deliberation is supposed to be more general than navigation, such as action planning or reasoning about action, then the need arises to sense more generally the recent relevant part of the world state and update its symbolic representation based on these sensor data. We call this representation the current *situation.*

The naive version of the update problem "Tell me all that is currently true about the world!" needs not be solved, luckily, if the goal is to build a concrete robot to work on a concrete task. *Only those facts need updating that, according to the symbolic domain model used for deliberation, are relevant for the robot to work on its task.* Then, every robot has its *sensor horizon,* i.e., a border in space and time limiting its sensor range. The term sensor is understood in a broad sense: It includes technical sensors like laser scanners, ultra sound transducers, or cameras; but if, for example, the arena of a delivery robot includes access to the control of an elevator, then a status request by wireless Ethernet to determine the current location of the elevator cabin is a sensor action, and the elevator status is permanently within the sensor horizon. We assume: *The world state information within the sensor horizon is sufficient to achieve satisfying robot performance.*

This said, the task of keeping the facts of a situation up-to-date remains to continually compute from recent sensor data and the previous situation a new version of the situation as far as it lies within the sensor horizon. The computation is based on plain, current sensor values as well as histories of situations and sensor readings or aggregates thereof. Practically, we cannot expect to get accurate situation updates instantly; all we can do is make the situation update as recent, comprehensive, and accurate as possible.

This paper contributes to this task an approach of using histories of activation values in behavior-based robot control systems (BBSs) [Mat99] as a *main* source of information for situation update. This approach is useful for three reasons:

- Activation values are calculated anyway in most BBSs to allow for arbitration or merging between behaviors; they can be used at no additional computation cost, provided that they are sufficiently fine grained.
- An activation value is grounded in sensor readings and, by definition, evaluates them in a way tailored to its respective behavior; using activation values like aggregated sensor readings yields automatically an action-centered way of "looking through the sensors".
- To have a practical hybrid robot control, the symbolic world model must be in accord with the inventory of behaviors anyway; using activation value histories in situation update only makes even more explicit the need to co-design the BBS and deliberative control components.

Our approach is not in principle limited to a particular combination of deliberation component and BBS, as long as the BBS is expressed as a dynamical system and involves a looping computation of activation values for the behaviors. Some additional requirements apply that will be clarified in the paper.

All demo examples are formulated in a concrete BBS framework, namely, Dual Dynamics (DD, [JC97]), which has also inspired our abstract view of BBSs. Our view of building hybrid robot controllers involving a BBS as reactive component is shaped by our work in progress on the DD&P robot control architecture [HJZM98,HS01], which blends DD controllers with action plans generated by a classical propositional planner (concretely, IPP [KNHD97]) as the central deliberation component. Mind, however: The method for fact extraction from activation value histories of BBSs presented here is potentially applicable in BBS frameworks other than DD, as will be discussed at the end of the paper.

The rest of this paper is organized as follows. In Sec. 2, we present our approach of formulating BBSs as dynamical systems and give a detailed example in Sec. 3. To provide some background concerning complete robot control systems, we then (Sec. 4) sketch how we assume the deliberation component interferes the BBS control component. Sec. 5 contains the technical contribution of the paper, describing in general as well as by way of example the technique of extracting facts from BBS activation value histories. Sec. 6 discusses the approach and relates it to the literature. Sec. 7 concludes.

2 BBSs as Dynamical Systems

We assume a BBS consists of two kinds of behaviors: low-level behaviors (LLBs), which are directly connected to the robot actuators, and higher-level behaviors (HLBs), which are connected to LLBs and/or HLBs. Each LLB implements two distinct functions: a target function and an activation function. The target function for the behavior b provides the reference t_b for the robot actuators (*"what to do"*) as follows:

$$t_b = f_b(s^T, s_f^T, \alpha_{LLB}^T) \tag{1}$$

where f_b is a nonlinear vector function with one component for each actuator variable, s^T is the vector of all inputs from sensors, s_f^T is the vector of the sensor-filters and α_{LLB}^T is the vector of activation values of the LLBs. By sensor-filters – sometimes called virtual sensors – we mean *markovian* and *non-markovian* functions used for processing specific information from sensors.

The LLB activation function *modulates* the output of the target function. It provides a value between 1 and 0, meaning that the behavior fully influences, does not influence or influences to some degree the robot actuators. It describes *when* to activate a behavior. For LLB b the activation value is computed from the following *differential equation*:

$$\dot{\alpha}_{b,LLB} = g_b(\alpha_{b,LLB}, OnF_b, OffF_b, OCT_b) \tag{2}$$

Eq. 2 gives the variation of the *activation value* $\alpha_{b,LLB}$ of this LLB. g_b is a nonlinear function. OCT_b allows the planner to influence the activation values, see Sec. 4. The scalar variables OnF_b and $OffF_b$ are computed as follows:

$$OnF_b = u_b(s^T, s_f^T, \alpha_{LLB}^T, \alpha_{HLB}^T) \tag{3}$$

$$OffF_b = v_b(s^T, s_f^T, \alpha_{LLB}^T, \alpha_{HLB}^T) \tag{4}$$

where u_b and v_b are nonlinear functions. The variable OnF_b sums up all conditions which recommend activating the respective behavior (on forces) and $OffF_b$ stands for contradictory conditions to the respective behavior (off forces).

The HLBs implement only the activation function. They are allowed to modulate only the LLBs or other HLBs on the same or *lower* level. In our case, the change of activation values for the HLBs $\alpha_{b,HLB}$ are computed in the same manner as Eq. 2.

The reason for updating behavior activation in the form of Eq. 2 is this. By referring to the previous activation value $\dot{\alpha}_b$, it incorporates a memory of the previous evolution which can be overwritten in case of sudden and relevant changes in the environment, but normally prevents activation values from exhibiting high-frequency oscillations or spikes. At the same time, this form of the activation function provides some *low-pass filtering* capabilities, deleting sensor noise or oscillating sensor readings.

Independent from that, it helps to develop stable robot controllers if behavior activations have a tendency of moving towards their boundary values, i.e., 0 or 1 in our formulation. To achieve that, we have implemented g_b in Eq. 2 as a *bistable* ground form (like in [BGG+99] for a RoboCup application of a BBS of the same type) providing some *hysteresis effect*. Without further influence, this function pushes activation values lower/higher than some threshold β (typically $\beta = 0.5$) softly to 0/1. The activation value changes as a result of *adding* the effects coming from the variables OnF, $OffF$, OCT and the bistable ground form. Exact formulations of the g_b function are then just technical and unimportant for this paper.

The relative smoothness of activation values achieved by using differential equations and bistability will be helpful later in the technical contribution of this paper (Sec. 5), when it comes to derive facts from the time series of activation values of the behaviors.

In our BBS formulation, behavior arbitration is achieved using the activation values. As shown in Eqs. 2 - 4, each behavior can interact with (i.e., encourage or inhibit) every other behavior on the same or lower level. The model of interaction between behaviors is defined by the variables OnF and $OffF$.

The output vector or *reference vector* r of the BBS for the robot actuators is generated by summing all LLB outputs by a *mixer*, as follows:

$$r = \sum_b \alpha_{b,LLB} t_b \tag{5}$$

Together with the form of the activation values, this way of blending the outputs of LLBs avoids *discontinuities* in the reference values r_i for the single robots actuators, such as sudden changes from full speed forward to full speed backward.

3 An Example

To illustrate the notation of Sec. 2 we give a demonstration problem consisting of the task of following a wall with a robot and entering only those doors that are wide enough to allow the entrance. Figure 1 gives an overview about the main part of our arena. The depicted robot is equipped with a short distance laser-scanner, 4 infrared side-sensors, 4 front/back bumpers and some dead-reckoning capabilities.

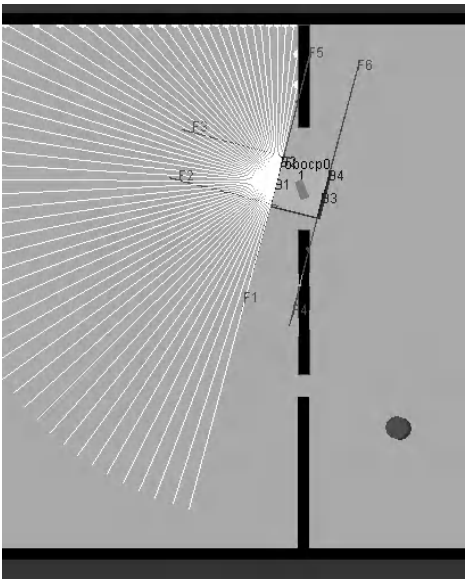


Fig. 1. The demo arena and the final robot pose in Example 1. The robot has started its course at the lower right corner, below to the round obstacle.

We used a simulator based on the DDDesigner prototype tool [Bre00,BGG+99]. The tool allows checking isolated behaviors or the whole BBS in designated environmental situations (configurations).

The control system contains three HLBs and six LLBs, see Fig. 2. $RobotDirection$ and $RobotVelocity$ are the references for the two respective actuators. We have the following HLBs (cf. Fig. 2):

CloseToDoor is activated if there is evidence for a door;
InCorridor is active while the robot moves inside a corridor;
TimeOut was implemented in order to avoid getting stuck in a situation, see Sec. 5;

The LLBs are the following:

TurnToDoor is activated if the robot is situated on a level with a door;
GoThruDoor is activated after the behavior TurnToDoor was successful;
FollowRightWall is active when a right wall is followed;
FollowLeftWall is active when a left wall is followed;
AvoidColl is active when there is an obstacle in the front of the robot
Wander is active when no other LLB is active.

Most of the implemented behaviors are common for this kind of tasks. However, we decided to split the task of passing a door in a sequence of two LLBs. This helps structure, maintain and independently improve these two behaviors.

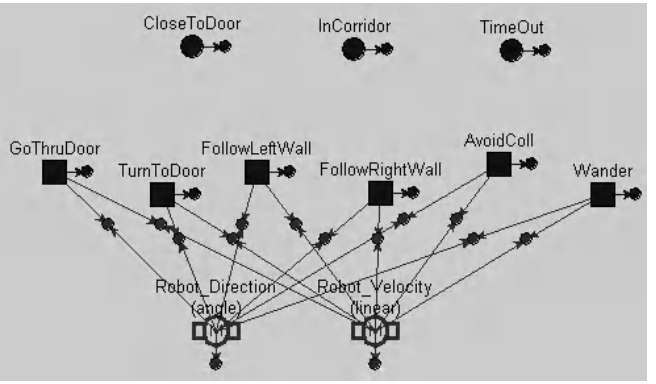


Fig. 2. The behavior inventory for our examples as in a screen shot from the DDDesigner tool. Big circles denote HLBs; squares denote LLBs; the hollow icons at the bottom denote robot actuator reference values. Arrows denote control flow between LLBs and actuators. The influence structure between behavior activations is not shown; the small circles are of no importance here.

To give a simple example of BBS modeling, here are the "internals" of Wander:

$$OnF_{\text{Wander}} = k_1(1-\alpha_{\text{CloseToDoor}}) * (1-\alpha_{\text{FollowRightWall}}) * (1-\alpha_{\text{FollowLeftWall}}) * (1-\alpha_{\text{AvoidColl}}) \tag{6}$$

$$OffF_{\text{Wander}} = k_2\alpha_{\text{AvoidColl}} + k_3\alpha_{\text{CloseToDoor}} + k_4\alpha_{\text{FollowRightWall}} + k_5\alpha_{\text{FollowLeftWall}} \tag{7}$$

$$RobotDirection_{\text{Wander}} = randomDirection() \tag{8}$$

$$RobotVelocity_{\text{Wander}} = mediumSpeed \tag{9}$$

$$t_{\text{Wander}} = \begin{bmatrix} RobotDirection_{\text{Wander}} \\ RobotVelocity_{\text{Wander}} \end{bmatrix} \tag{10}$$

$$\dot{\alpha}_{\text{Wander}} = g_{\text{Wander}}(\alpha_{\text{Wander}}, OnF_{\text{Wdr}}, OffF_{\text{Wdr}}, OCT_{\text{Wdr}}) \tag{11}$$

where $k_1 \ldots k_5$ are empirically chosen constants. $randomDirection()$ could be every function that generates a direction which results in a randomly chosen trajectory.

Due to its *product* form, OnF_{Wander} can only be remarkably greater than zero if all included α_b are approximately zero. $OffF_{\text{Wander}}$ consists of a *sum* of terms allowing every included behavior to deactivate Wander. Both terms are simple and can be calculated extremely fast, which is a guideline for most BBSs. The OCT term will be briefly explained in the next section.

Fig. 3 shows the *activation value histories* generated during a robot run, which will be referred to as Example 1. The robot starts at the right lower edge of Fig. 1 with Wander in control for a very short time, until a wall is perceived. This effect is explained by Eq. 6. While the robot starts to follow the wall, it detects the small round obstacle in front. In consequence, two LLBs are active simultaneously: AvoidColl and FollowLeftWall. Finally, the robot follows the wall, ignores the little gap and enters the door. In the

examples for this paper, FollowRightWall is always inactive and therefore not shown in the activation value curves.

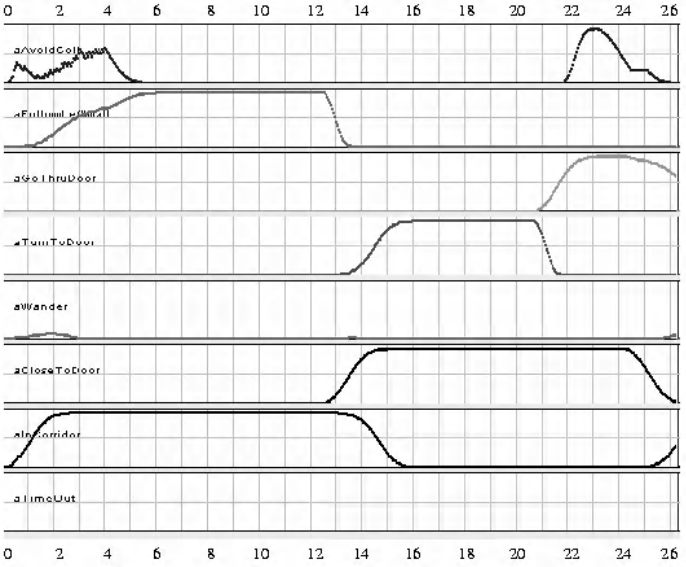


Fig. 3. The activation value histories for Example 1. The numbers are time unit for reference.

This exemplifies the purpose of a *slow* increase in behavior activation. FollowLeftWall should only have a strong influence to the overall robot behavior if a wall is perceived with *both* side-sensors for some time, so as to be more sure that the robot really has sensed a wall. The small dent in the activation of FollowLeftWall (around the time $t = 4$) is explained by perceiving free space with one side-sensor. If both side-sensors detect free space this behavior would be deactivated. The turning to the door is described by rising/falling edges of some activation values. The second rise of AvoidColl (after $t = 22$) is caused by the door frame, which pops into sight as a close obstacle at the very end of the turning maneuver. Effectively, the collision avoidance guides the robot through the door. Finally GoThruDoor gets slowly deactivated allowing other behaviors to take control of the robot.

The HLBs CloseToDoor and InCorridor describe global states, thereby modulating the interaction, activation and sequencing of the LLBs.

4 From Plans to BBSs: Blending Behaviors with Operators

The technical contribution of this paper is an approach to enhancing the information flow from the BBS to the deliberation part in hybrid robot control systems. Before coming to that in Section 5, we want to sketch the control flow in the opposite direction, from the

planner to the BBS, to make complete the picture of the entire robot control architecture that we have in mind. Not in the focus of the present paper, this description just consists of stating the basic principle, and we refer to work on the DD&P control architecture [HJZM98,HS01], which elaborates on the approach.

The basic idea is this: An action planner continually maintains a current action plan, based on the current situation and the current set of user-provided or self-generated mission goals. Based on the current plan and the current situation, an execution component picks one of the operators in the plan as the one currently to be executed. Plan execution is done in a *plans-as-advice* [Pol92] fashion: Executing an operator means stimulating more or less strongly the behaviors working in favor of the operator, and muting those working against its purpose. Which operator stimulates or mutes which behaviors is an information that the domain modeler has to provide along with the domain model for the deliberative component and the set of behaviors for the BBS.

Technically, the influence of the current operator is "injected" into the BBS in terms of the *Operator-Coupling-Terms* (OCT) in the activation functions, see Eqs. 2. The influence of the current ground operator op gets inside every behavior b through the term OCT_b, as follows:

$$OCT_b = \sum_{op \in OP} s_b^{op} c_b^{op} (Z_b^{op} - \alpha_b) \tag{12}$$

where $s_b^{op}, Z_b^{op} \in \{0, 1\}$ and c_b^{op} is a constant. $s_b^{op} = 1$ iff op influences the behavior b. c_b^{op} models the immediacy or delay of the operator influence on the behavior. Z_b^{op} expresses whether the operator influence is of the stimulating or the muting sort: If $Z_b^{op} = 1$, then the respective behavior is stimulated, and muted if $Z_b^{op} = 0$. Z may be a boolean function, returning 0 or 1 conditionally.

To give an example, assume that the domain model for the deliberation component includes an operator **GO-IN-RM**(x) modeling the action of some office delivery robot to go (from wherever it is) to and enter room x. Let the behavior inventory be the one specified in Sec. 3. Here is a selection of s, Z, and c variables of these behaviors and how they should be affected by the ground operator **GO-IN-RM**(A) :

$s_{\mathsf{GoThruDoor}}^{\textbf{GO-IN-RM}(A)}$ set to 1 as the operator does influence the behavior;
$c_{\mathsf{GoThruDoor}}^{\textbf{GO-IN-RM}(A)}$ set to some medium value, causing a tendency to influence activation soon after the operator is chosen as being active;
$Z_{\mathsf{GoThruDoor}}^{\textbf{GO-IN-RM}(A)}$ set to *charFct*($\mathsf{CloseTo}(A)$), i.e., the characteristic function that returns 1 if $\mathsf{CloseTo}(A)$ is currently in the fact base, and 0 else;
$s_{\mathsf{Wander}}^{\textbf{GO-IN-RM}(A)}$ set to 1 as the operator should affect its activation (namely, muting it);
$s_{\mathsf{AvoidColl}}^{\textbf{GO-IN-RM}(A)}$ set to 0 as collision avoidance should not be affected by the operator (note the difference between not affecting and actively muting an activation value)

5 From BBSs to Plans: Extracting Facts from Activation Values

We now turn to the method how to extract facts from activation value histories. It is influenced by previous work on chronicle recognition, such as [Gha96].

To start, take another look at the activation curves in Fig. 3 in Sec. 3. Some irregular activation time series occur due to the dynamics of the robot/environment interaction,

such as early in the AvoidColl and Wander behaviors. However, certain patterns reoccur for single behaviors within intervals of time, such as a value being more or less constantly high or low, and values going up from low to high or vice versa. The idea to extract symbolic facts from activation values is to consider characteristic groups or *gestalts* of such qualitative activation features occurring in *chronicles* over time.

To make this precise, we define, first, *qualitative activation values* (or briefly, qualitative activations) describing these isolated patterns. In this paper, we consider four of them, which are sufficient for defining and demonstrating the principle, namely, rising/falling edge, high and low, symbolized by predicates $\Uparrow$e, $\Downarrow$e, Hi, and Lo, respectively. In general, there may be more qualitative activations of interest, such as a value staying in a medium range over some period of time. For a behavior b and time interval $[t_1, t_2]$, they are defined as

$$\begin{aligned}
\mathsf{Hi}(b)[t_1,t_2] &\equiv \alpha_b[t] \geq h \text{ for all } t_1 \leq t \leq t_2 \\
\mathsf{Lo}(b)[t_1,t_2] &\equiv \alpha_b[t] \leq l \text{ for all } t_1 \leq t \leq t_2 \\
\Uparrow\mathsf{e}(b)[t_1,t_2] &\equiv \alpha_b[t_1] = l \text{ and } \alpha_b[t_2] = h \text{ and} \qquad (13)\\
&\quad\ \alpha_b \text{ increases } \textit{generally monotonically} \text{ over } [t_1,t_2] \\
\Downarrow\mathsf{e}(b)[t_1,t_2] &\equiv \alpha_b[t_1] = h \text{ and } \alpha_b[t_2] = l \text{ and} \qquad (14)\\
&\quad\ \alpha_b \text{ decreases } \textit{generally monotonically} \text{ over } [t_1,t_2]
\end{aligned}$$

for given threshold values $0 \ll h \leq 1$ and $0 \leq l \ll 1$, where $\alpha_b[t]$ denotes the value of α_b at time t. *General monotonicity* requires another technical definition, which we skip here for brevity. The idea is that some degree of noise should be allowed in, e.g., an increasing edge, making the increase locally non-monotonic. In the rather benign example activation curves in this paper, regular monotonicity suffices. Similarly, it is not always reasonable to use the global constants h, l as Hi and Lo thresholds, respectively. It is possible to use different threshold constants or thresholding functions for different behaviors. We do not go into that here. Then, it makes sense to require a minimum duration for $[t_1, t_2]$ to prevent useless mini intervals of Hi and Lo types from being identified. Finally, the strict equalities in Eq.s 13 and 14 are unrealistic in real robot applications, where two real numbers must be compared, which are seldom strictly equal. Equality $\pm\epsilon$ is the solution of choice here.

The key idea to extract facts from activation histories is to consider patterns of qualitative activations of several behaviors that occur within the same interval of time. We call these patterns *activation gestalts*. We express them formally by a time-dependent predicate AG over a set Q of qualitative activations of potentially many behaviors. For a time interval $[t, t']$ the truth of $AG(Q)[t, t']$ is defined as the conjunction of conditions on the component qualitative activations $q \in Q$ of behaviors b in the following way:

$$\begin{aligned}
&\text{case } q = \mathsf{Hi}(b) \text{ then } \mathsf{Hi}(b)[t,t'] \\
&\text{case } q = \mathsf{Lo}(b) \text{ then } \mathsf{Lo}(b)[t,t'] \\
&\text{case } q = \Uparrow\mathsf{e}(b) \text{ then } \Uparrow\mathsf{e}(b)[t_1,t_2] \text{ for some } [t_1,t_2] \subseteq [t,t'], \text{ and } \mathsf{Hi}(b)[t_2,t'] \\
&\text{case } q = \Downarrow\mathsf{e}(b) \text{ then } \Downarrow\mathsf{e}(b)[t_1,t_2] \text{ for some } [t_1,t_2] \subseteq [t,t'], \text{ and } \mathsf{Lo}(b)[t_2,t']
\end{aligned}$$

Note that it is not required that different rising or falling edges in Q start or end synchronously among each other or at the interval borders of $[t, t']$—they only must all occur somewhere within that interval.

For example, $AG(\{\Uparrow\mathsf{e}(\mathsf{GoThruDoor}), \Downarrow\mathsf{e}(\mathsf{TurnToDoor}), \mathsf{Hi}(\mathsf{CloseToDoor})\})$ is true over $[20, 24]$ in the activation histories in Fig. 3; it is also true over $[16, 23]$ (and therefore, also over their union $[16, 24]$), but not over $[16, 25]$, as CloseToDoor has left its Hi band by time 25, and possibly the same for GoThruDoor, depending on the concrete value of the h threshold.

A *chronicle* over some interval of time $[t_0, t]$ is a set of activation gestalts over subintervals of $[t_0, t]$ with a finite set of n linearly ordered internal *interval boundary points* $t_0 < t_1 < \cdots < t_n < t$. A ground fact is extracted from the activation history of a BBS as true (or rather, as *evident*, see the discussion below) at time t if its *defining chronicle* has been observed over some interval of time ending at t. The defining chronicle must be provided by the domain modeler, of course.

We give as an example the defining chronicle of the fact InRoom that the robot is in some room, such as the one left of the wall in Fig. 1. $\mathsf{InRoom}[t]$ is extracted if the following defining chronicle is true within the interval $[t_0, t]$, where the t_i are existentially quantified:

$$\begin{aligned}
& AG(\{\Downarrow\mathsf{e}(\mathsf{GoThruDoor})\})[t_4, t] \\
\wedge\ & AG(\{\Uparrow\mathsf{e}(\mathsf{GoThruDoor}), \Downarrow\mathsf{e}(\mathsf{TurnToDoor}), \mathsf{Hi}(\mathsf{CloseToDoor})\})[t_3, t_4] \\
\wedge\ & AG(\{\mathsf{Hi}(\mathsf{TurnToDoor}, \mathsf{Lo}(\mathsf{InCorridor})\})[t_2, t_3] \\
\wedge\ & AG(\{\Uparrow\mathsf{e}(\mathsf{TurnToDoor}), \Uparrow\mathsf{e}(\mathsf{CloseToDoor}), \Downarrow\mathsf{e}(\mathsf{InCorridor})\})[t_1, t_2] \\
\wedge\ & AG(\{\mathsf{Hi}(\mathsf{InCorridor})\})[t_0, t_1] \\
\wedge\ & AG(\{\mathsf{Lo}(\mathsf{TimeOut})\})[t_0, t]
\end{aligned} \tag{15}$$

Assuming reasonable settings of the Hi and Lo thresholds h, l, the following substitutions of the time variables to time-points yield the mapping into the activation histories in Fig. 3: $t = 28$ (right outside the figure), $t_0 = 3, t_1 = 12, t_2 = 16, t_3 = 20, t_4 = 24$. As a result, we extract $\mathsf{InRoom}[24]$.

This substitution is not unique. For example, postponing t_0 until 5 or having t_1 earlier at 9 would also work. This point leads to the process of *chronicle recognition:* given a working BBS, permanently producing activation values, how are the given defining chronicles of facts checked against that activation value data stream to determine whether some fact starts to hold?

The obvious basis for doing this is to keep track of the qualitative activations as they emerge. That means, for every behavior, there is a process logging permanently the qualitative activations. For those of type Hi and Lo, the sufficiently long time periods of the respective behavior activation above and below the h, l thresholds, resp., have to be recorded and, if adjacent to the current time point, appropriately extended. This would lead automatically to identifying qualitative activations of types Hi and Lo with their earliest start point, such as $t_0 = 3$ for $\mathsf{Hi}(\mathsf{InCorridor})$ in the example above. Qualitative activations of types $\Uparrow\mathsf{e}$ and $\Downarrow\mathsf{e}$ are logged iff their definitions (eqs. 13 and 14, resp.) are fulfilled in the recent history of activation values. As this logging process is local to every behavior, the complexity is linear $O(B)$ in the number B of behaviors.

Qualitative activation logs are then permanently analyzed whether any of the existing defining chronicles are fulfilled, which may run in parallel to the ongoing process of

logging the qualitative activations. An online version of this analysis inspired by [Gha96] would attempt to match the flow of qualitative activations with all defining chronicles c by means of *matching fronts* that jump along c's internal interval boundary points t_i and try to bind the next time point t_{i+1} as current matching front such that the recent qualitative activations fit all sub-intervals of c that end in t_{i+1}. Note that more than one matching front may be active in every defining chronicle at any time. A matching front in c vanishes if it reaches the end point t (the defining chronicle is true), or else while stuck at t_i is caught up by another matching front at t_i, or else an activation gestalt over an interval ending at t_{i+1} is no longer valid in the current qualitative activation history.

For complexity considerations, assume that C defining chronicles are defined that involve a maximum of $N-1$ internal interval boundary points. Assume further that A is the maximal number of qualitative activations occurring in single activation gestalt conjuncts of all defining chronicles. (A is bounded by the number B of behaviors.) Then, one cycle of the online matching of defining chronicles runs in $O(ACN)$ time.

Practically, the necessary computation may be focused by specifying for each defining chronicle a *trigger condition,* i.e., one of the qualitative activations in the definition that is used to start a monitoring process of the validity of all activation gestalts. For example, in the InRoom definition above, $\Uparrow$e(GoThruDoor), as occurring in the $[t_3, t_4]$ interval, might be used. Note that the trigger condition need not be part of the earliest activation gestalts in the definition. On appearance of some trigger condition in the qualitative activation log, we try to match the activation gestalts prior to the trigger with qualitative activations in the log file, and, if successful, verify the gestalts after the trigger condition in the qualitative activations as they are being logged.

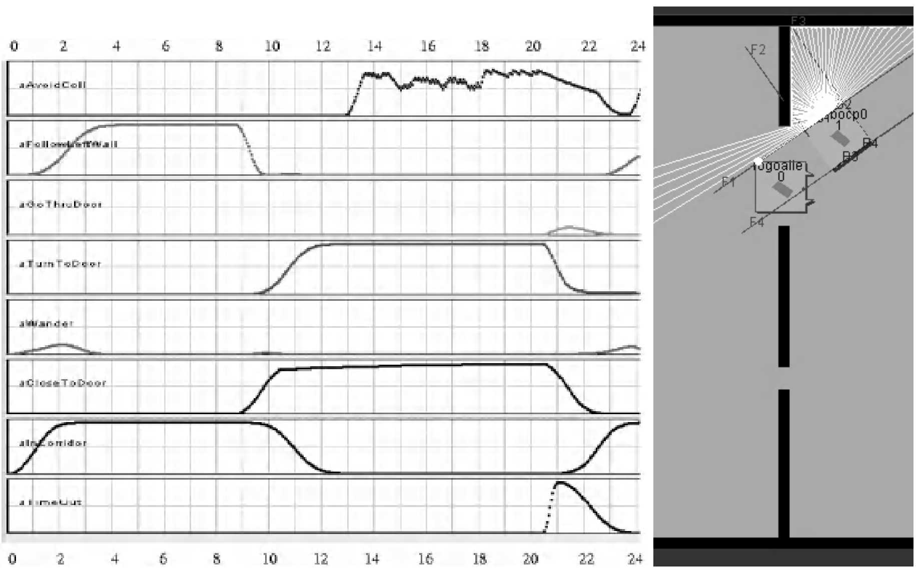


Fig. 4. Robot example 2: final state and activation curves. See text for explanations.

To give an example where the derivation of the InRoom fact fails, consider Fig. 4. The scenario is like before, i.e., the robot starts at the lower right corner, driving upward and trying to enter any door large enough. Different to Fig. 1, no obstacle is present at the beginning, and while the robot tries to enter into the detected door, another robot comes from within the room and blocks it. Fig. 4 right shows the scene after the robot has failed to enter the door, and left are the respective activation values.

Like before, while InCorridor ($t_0 = 3$), the CloseToDoor and TurnToDoor activations rise with InCorridor falling ($t_1 = 9, t_2 = 13$); then, TurnToDoor is Hi, while InCorridor is Lo ($t_3 = 20$). But then, mischief strikes. After the long high period of TurnToDoor, TimeOut jumps up, terminating its Lo period, before GoThruDoor has risen. In consequence, t_4 cannot be bound, and the fact InRoom not extracted. If $\Uparrow$e(GoThruDoor) was used as a trigger condition in the first place, then no unnecessary matching effort was wasted.

Some more general remarks are in place here. Defining chronicles exclusively in terms of activation gestalts is a special case that we have used in this paper to keep matters focused. In general, the obvious other elements may be used for defining them: sensor readings at some time points (be they physical sensors or sensor filters), and the validity of symbolic facts at a time point or over some time interval. Our intention is to provide fact extraction from activation values as a *main* source of information, not the exclusive one. That type of information can be added to the logical format of a chronicle definition as in (15). For example, if the exact time point of entering a room with the robot's front is desired as the starting point of the InRoom fact, then this might be determined by the time within the interval $[t_4, t]$ (i.e., within the decrease of the GoThruDoor activation) where some sensor senses open space to the left and right again. As another example, assume that the fact At($Door_A$) for the door to some room A may be in the fact base (as derived from a normal localization process). Then At($Door_A$)$[t_4]$ could be added to the defining chronicle (15) above to derive not only InRoom$[t]$, but more specifically InRoom$(A)[t]$.

The fact extraction technique does not presume or guarantee anything about the consistency of the facts that get derived over time. Achieving and maintaining consistency, and determining the ramifications of newly emerged facts remain issues that go beyond fact extraction. Pragmatically, we would not recommend to blindly add a fact as true to the fact base as soon as its defining chronicle has been observed. A consequent of a recognized defining chronicle should be interpreted as evidence for the fact or as a fact *hypothesis,* which should be added to the robot's knowledge base only by a more comprehensive knowledge base update process, which may even reject the hypothesis in case of conflicting information. A possible solutions would be to add some integrity constraints to the defining chronicles. However, this is not within the scope of this paper.

6 Discussion

A physical agent's perception categories must to some degree be in harmony with its actuator capabilities—at least in purposively designed technical artifacts such as working

autonomous robots.[1] Our approach of extracting symbolic facts from behavior activation merely exploits this harmony for intertwining control on a symbolic and a reactive level of a hybrid robot control architecture.

The technical basis for the exploitation are time series of behavior activation values. We have taken them from a special type of behavior-based robot control systems (BBSs), namely, those consisting of behaviors expressed by nonlinear dynamical functions of a particular form, as described in Sec. 2. The point of having activation values in BBSs is not new; it is also the case, e.g., for the behavior-based fuzzy control part underlying Saphira [KMSR97], where the activation values are used for context-dependent *blending* of behavior outputs, which is similar to their use in our BBS framework. Activation values also provide the degree of applicability of the corresponding motor schemas in [Ark98, p. 141].

The activation values of a dynamical system-type BBS are well-suited for fact extraction in that their formal background in dynamical systems theory provides both the motivation and the mathematical inventory to make them change smoothly over time—compare, e.g., the curves in Figures 3 and 4 with the ragged ones in [SRK99, Fig. 5.10]. This typical smoothness is handy for defining *qualitative activations,* which aggregate particular patterns in terms of edges and levels of the curves of individual behaviors, which are recorded as they emerge over time. These then serve as a stable basis for chronicle recognition over qualitative activations of several behaviors. Note, however, that this smoothness is a practical rather than a theoretical issue, and other BBS approaches may serve as bases for fact extraction from activation values.

We want to emphasize that the activation values serve two purposes in our case: first, their normal one to provide a reliable BBS, and second, to deliver the basis for extracting persistent facts, based on their distinctive patterns. With the second use, we save the domain modeler a significant part of the burden of designing a complicated sensor interpretation scheme only for deriving facts. The behavior activation curves, as a by-product coming for free of the behavior-based robot control, focus on the environment dynamics, be it induced by the robot itself or externally. By construction, these curves aggregate the available sensor data in a way that is particularly relevant for robot action. We have argued that this information can be used as a main source of information about the environment; other information, such as coming from raw sensor data, from dedicated sensor interpretation processes, or from available symbolic knowledge, could be used in addition.

As activation values are present in a BBS anyway, it is possible to "plug-in" the fact extraction machine for a deliberative component to an already existing behavior system like the DD control system in [BGG+99]. Yet, if a new robot control system is about to be written for a new application area, things could be done better, within the degrees of freedom for variations in behavior and domain model design. The ideal case is that the behavior inventory and the fact set is in harmony in the sense that such facts get used in the domain model whose momentary validity engraves itself in the activation value history, and such behaviors get used that produce activation values producing evidence for facts. For example, a single WallFollow behavior working for walls on the right and

[1] We do not speculate about biological agents in this paper, although we would conjecture that natural selection and parsimony strongly favor this principle.

on the left, may be satisfactory from the viewpoint of behavior design for a given robot application; for fact extraction, it may be more opportune to split it into FollowLeftWall and FollowRightWall, which would be equally feasible for the behavior control, but allows more targeted facts to be deduced directly.

Apart from such design-level interdependencies, which are non-trivial, but not special for our approach, we are aiming at a control architecture with a deliberative and a behavior-based part as two abreast modules with no hierarchy, as sketched in [HJZM98]. The fact extraction scheme leaves the possibility to un-plug the deliberative part from the robot control, which we think is essential for robustness of the whole robot system.

Our technique is complementary to anchoring symbols to sensor data as described in [CS01]. It differs from that line of work in two main respects. First, we use sensor data as aggregated in activation value histories only, not raw sensor data. Second, we aim at extracting ground facts rather than establishing a correspondence between percepts and references to physical objects. The limit of our approach is that it is inherently robot-centered in the sense that we can only arrive at information that has to do directly with the robot action. The advantage is that, due to its specificity, it is conceptually and algorithmically simpler than symbol anchoring in general.

7 Conclusion

We have presented a new approach for extracting information about symbolic facts from activation curves in behavior-based robot control systems. Updating the symbolic environment situation is a crucial issue in hybrid robot control architectures in order to bring to bear the reasoning capabilities of the deliberative control part on the physical robot action as exerted by the reactive part. Unlike standard approaches to sensing the environment in robotics, we are using the information hidden in the temporal development of the data, rather than their momentary values. Therefore, our method promises to yield environment information that is complementary to normal sensor interpretation techniques, which can and should be used in addition.

We have presented the technique in principle as well as in terms of selected demo examples in a robot simulator, which has allowed to judge the approach feasible and to design the respective algorithms. The computational complexity of the recognition process is in $O(BCN)$, where B is the number of behaviors, C the number of chronicle definitions, and N the maximal “length" of a chronicle definition in terms of intermediate time points internal to the chronicle definition.

The approach will be applied in the context of the hybrid robot control architecture DD&P [HJZM98] to generate an important part of the information that is used to update the symbolic world model from the sensor data stream. Work is ongoing towards a physically concurrent implementation of DD&P on physical robots, as described in [HS01].

Acknowledgements. Thanks to Herbert Jaeger, the GMD AiS.BE-Team, and especially Hans-Ulrich Kobialka for their support on this work.

References

[Ark98] R. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, 1998.

[BFG+97] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *J. Expt. Theor. Artif. Intell.*, 9:237–256, 1997.

[BGG+99] A. Bredenfeld, W. Göhring, H. Günter, H. Jaeger, H.-U. Kobialka, P.-G. Plöger, P. Schöll, A. Siegberg, A. Streit, C. Verbeek, and J. Wilberg. Behavior engineering with *dual dynamics* models and design tools. In *Proc. 3rd Int. Workshop on RoboCup at IJCAI-99*, pages 57–62, 1999.

[Bre00] A. Bredenfeld. Integration and evolution of model-based tool prototypes. In *11th IEEE International Workshop on Rapid System Prototyping*, pages 142–147, Paris, France, June 2000.

[CS01] S. Coradeschi and A. Saffiotti. Perceptual anchoring of symbols for action. In *Proc. of the 17th IJCAI Conf.*, to appear, Seattle, WA, August 2001.
Online at http://www.aass.oru.se/~asaffio/.

[FBT99] D. Fox, W. Burgard, and S. Thrun. Markov localization for mobile robots in dynamic environments. *J. Artif. Intell. Research (JAIR)*, 11, 1999.

[Gha96] M. Ghallab. On chronicles: Representation, on-line recognition and learning. In Aiello, Doyle, and Shapiro, editors, *Proc. Principles of Knowledge Representation and Reasoning (KR-96)*, pages 597–606. Morgan-Kauffman, November 1996.

[HJZM98] J. Hertzberg, H. Jaeger, U. Zimmer, and Ph. Morignot. A framework for plan execution in behavior-based robots. In *Proc. of the 1998 IEEE Int. Symp. on Intell. Control (ISIC-98)*, pages 8–13, Gaithersburg, MD, September 1998.

[HS01] J. Hertzberg and F. Schönherr. Concurrency in the DD&P robot control architecture. In M. F. Sebaaly, editor, *Proc. of The Int. NAISO Congress on Information Science Innovations (ISI'2001)*, pages 1079–1085. ICSC Academic Press, march, 17–21 2001.

[JC97] H. Jaeger and Th. Christaller. Dual dynamics: Designing behavior systems for autonomous robots. In S. Fujimura and M. Sugisaka, editors, *Proc. Int. Symposium on Artificial Life and Robotics (AROB'97)*, pages 76–79, 1997.

[KMSR97] K. Konolige, K. Myers, A. Saffiotti, and E. Ruspini. The Saphira architecture: A design for autonomy. *J. Expt. Theor. Artif. Intell.*, 9:215–235, 1997.

[KNHD97] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning. 4th Eur. Conf. on Planning, ECP-97*, pages 273–285. Springer (LNAI 1348), 1997.

[Mat99] M. J. Matarić. Behavior-based robotics. In Robert A. Wilson and Frank C. Keil, editors, *MIT Encyclopedia of Cognitive Sciences*, pages 74–77. The MIT Press, Cambridge, Massachusetts, April 1999.

[MNPW98] N. Muscettola, P. Nayak, B. Pell, and B.C. Williams. Remote Agent: to boldly go where no AI system has gone before. *J. Artificial Intelligence*, 103:5–47, 1998.

[Pol92] M.E. Pollack. The uses of plans. *Artif. Intell.*, 57(1):43–68, 1992.

[Rem00] Remote Agent Project. Remote agent experiment validation.
http://rax.arc.nasa.gov/DS1-Tech-report.pdf, February 2000.

[SRK99] A. Saffiotti, E.H. Ruspini, and K. Konolige. Using fuzzy logic for mobile robot control. In H-J. Zimmermann, editor, *Practical Applications of Fuzzy Technologies*, chapter 5, pages 185–205. Kluwer Academic, 1999. Handbook of Fuzzy Sets, vol.6.

Learning Search Control Knowledge for Equational Theorem Proving

Stephan Schulz

Institut für Informatik, Technische Universität München,
D-80290 München, Germany, schulz@informatik.tu-muenchen.de

Abstract. One of the major problems in clausal theorem proving is the control of the proof search. In the presence of equality, this problem is particularly hard, since nearly all state-of-the-art systems perform the proof search by saturating a mostly unstructured set of clauses. We describe an approach that enables a superposition-based prover to pick good clauses for generating inferences based on experiences from previous successful proof searches for other problems. Information about good and bad search decisions (useful and superfluous clauses) is automatically collected from search protocols and represented in the form of *annotated clause patterns*. At run time, new clauses are compared with stored patterns and evaluated according to the associated information found. We describe our implementation of the system. Experimental results demonstrate that a learned heuristic significantly outperforms the conventional base strategy, especially in domains where enough training examples are available.

1 Introduction

The last few years have seen an impressive improvement in the power of first-order theorem provers, and a corresponding increase in the use these systems in research and development. They are being used for the verification of protocols [Sch97,Wei99,GP00], and the retrieval of mathematical theorems [DW97] or software components [FS97] from libraries. Theorem provers are also used to synthesize larger programs from standard building blocks and to prove the correctness of the resulting program systems [SWL+94,BRLP98]. For these purposes, various systems have been embedded into larger, interactive proof systems like ILF [DGHW97], KIV [RSS95] or VSE [HLS+96].

The most visible success of automatic theorem provers (or *ATP systems*) today is the celebrated proof of the Robbins algebra problem by EQP [McC97]. Successes like this demonstrate the power of current theorem proving technology. However, despite the fact that ATP systems are able to perform basic operations at an enormous rate, and can solve most simple problems much faster than any human expert, they still fail on many tasks routinely solved by mathematicians, even if these tasks can be encoded in first-order logic in a natural way. Moreover, many of the more impressive successes require an experienced human user who selects a suitable prover configuration, often by trial and error.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 320–334, 2001.

The main reason for this is that a theorem prover has to search for a proof in a usually infinite search space with a very high branching factor, i.e. a very high number of possible choices at each choice point. Much previous work in theorem proving has been targeted at the development of refined calculi that reduce this branching factor by restricting the number of possible inferences. However, the semi-decidability of the underlying problem for most interesting logics restricts the potential for this approach, and even the most refined calculi typically are highly non-deterministic. This is particularly true for calculi that deal with equality, as the congruence properties of this relation imply a very large number of possible inferences. Moreover, all calculi for equational logic that have been successfully implemented and show good performance operate on unstructured clause sets, and lack the inherent goal-orientedness of e.g. model elimination calculi or the set-of-support strategy for non-equational logic.

Most current theorem provers therefore use a small set of highly parameterized *heuristic evaluation functions* to guide the proof search. The selection of a proper evaluation function and set of parameters for a given problem (or problem domain) is based on experience of the human user, often supported by large and tedious sets of experiments. It is often encoded in a so-called *automatic mode*, where the prover selects one of a relatively small set of pre-configured strategies based on properties of the current proof problem. While such automatic configuration can improve the performance of a prover if used by a non-expert, the development of *new* strategies is still a manual process that requires a lot of work and attention by an expert user or even developer, and is hence very expensive in terms of man power.

In this paper we suggest a way to automatically *learn* information about good and bad search decisions from examples of successful proof searches, and to use the learned knowledge to define new search control heuristics. A proof search is represented by a clause dependency graph, and clauses contributing to a proof as well as superfluous clauses are identified. A suitable subset of these clauses is used to represent good and bad search decisions. We encode these clauses as *patterns* that abstract from irrelevant information and use the information about the value of a decision in later proof searches to compute an informed evaluation of new clauses.

For a broad overview of related works, see [DFGS99]. Our own previous work in this direction (see [DS96,DS00]) was restricted to the unit-equational case, where the suitable encoding of search decisions is much easier. Moreover, these previous approaches were based on positive search decisions only, and, due to the relatively small number of unit-equational proof problems publically available, we did not perform an experimental evaluation of the same rigidity. Other related work has been reported by Fuchs [Fuc96,Fuc97]. The main differences are that Fuchs again focuses on the unit-equational case, that his clause generalizations are weaker, and that he tries to replay search decisions from a single proof problem. Our approach, on the other hand, gathers experience from an arbitrary number of proof searches.

We start the remainder of this paper with a very short introduction to superposition-based theorem proving and the organization of the search process. In the next section we describe our approach to learning and cover knowledge acquisition, representation, and application. Section 4 contains experimental results, and we conclude in section 5.

2 Superposition-Based Theorem Proving

The superposition calculus [BG94,BG98] and its variants are generally recognized as the most powerful calculi currently available to tackle theorem proving problems with equality. The well-known prover SPASS [WAB+99] and our own system E [Sch99,Sch01] are based on particular instances of this calculus, and all other leading saturating theorem provers incorporate at least some important features of it. We will give a very short introduction, concentrating on aspects important for search control.

2.1 Calculus

The superposition calculus is a refutational calculus. It works on a set of clauses, and tries to show the unsatisfiability of the clause set by deriving the empty clause. A clause in this calculus is a multi-set of literals (positive or negative equations[1]) and is interpreted as the universally quantified disjunction of these equations. An equation is an unordered pair of first-order terms. Generating inferences add new clauses to the set, contracting inferences modify or remove existing clause. A theorem proving derivation is a sequence of generating and contracting inferences applied to a set of clauses. Most inference types are restricted by a *simplification ordering* on terms, which is extended to equations and clauses.

The most important generating inference in any superposition calculus is the superposition inference, a paramodulation inference restricted to maximal terms in maximal literals[2]. It can be seen as lazy conditional rewriting applied to instances of clauses resulting from unification.

Despite the restrictions imposed by ordering and literal selection, superposition inferences are typically responsible for more than 99% of all clauses generated during a proof search. In addition to superposition inferences, special factoring inferences and equality resolution inferences are necessary for the completeness of the calculus. However, the number of clauses generated by these inferences typically is miniscule.

One of the major strengths of the superposition calculus is its compatibility with a wide range of contracting inferences. Probably the most important of these contracting inferences is the *rewriting* of clauses i.e. the use of orientable

[1] Non-equational atoms are encoded as equations.

[2] In many cases, inferences can be further restricted by *literal selection*, i.e the restriction of paramodulation inferences to a single negative literal.

instances of positive unit clauses to replace terms with smaller terms. Other contracting inferences are the removal of trivial or duplicated literals. Tautological or subsumed clauses can be eagerly removed without affecting the completeness of a proof search (assuming a suitable definition of subsumption). These contracting processes typically take up a large part of the total work during a proof search, and are totally indispensable for the efficient work of the prover.

2.2 Organizing the Proof Search

The superposition calculus is complete, i.e. for any unsatisfiable clause set there is a theorem proving derivation that derives the empty clause. However, in order to guarantee that the empty clause is found, certain *fairness* constraints are imposed upon the search. A sufficient way to ensure that these constrains are met is to eventually perform all generating inferences between *persistent* clauses, i.e. to ensure that no possible inference is delayed forever. This is achieved by different variants of the *given-clause* algorithm. This algorithm uses two sets of clauses, a set of *unprocessed* clauses, which initially contains the input clauses, and a set of *processed clauses*, which initially is empty. The algorithm repeatedly picks one of the unprocessed clauses, performs all generating inferences between this *given clause* and all clauses in the processed set, adds the newly produced clauses into the set of unprocessed clauses, and moves the given clause to the set of processed clauses.

The selection of the given clause is typically done by a heuristic evaluation function which rates each clause. The most widely used and successful heuristics are based on symbol counting, and assign a low weight to clauses with a low number of symbols. This heuristic may be interleaved with a *first-in first-out* strategy that always picks the oldest remaining clause.

Different implementations of the given-clause algorithm differ mainly in the way in which contracting inferences are performed. In our version, the given clause is rewritten with the processed clause set and checked for subsumption after being picked. Then we purge clauses that can be subsumed by or rewritten with the given clause from the set of processed clauses. In addition, newly generated clauses are rewritten and checked for obvious tautologies. Other variants of the algorithm also keep the list of unprocessed clauses in normal form or use generated unit clauses more eagerly for rewriting processed clauses (backward contraction). We will abstract from contracting inferences in the following, for a discussion of their effect and handling see [Sch00].

3 Learning Search Control Knowledge

We have performed a detailed analysis of the given clause algorithm, supported by experimental data, and found that the selection of the given clause is the most critical choice point in this algorithm [Sch00]. In a typical proof search today, a high-performance prover generates up to a few million clauses within 5 minutes. Of these, up to about 10000 are processed. However, typically much

less than 200 clauses are needed for the proof. If we assume an optimal oracle, processing these 200 clauses only takes minimal time for modern theorem provers – typically less then a second, and in virtually all cases less than 5 seconds.

Based on these results, we consider the choice of the next clause to process the most crucial choice point in the given-clause algorithm. We try to improve the proof search by learning good evaluations for clauses based on experiences from previous proof searches.

In order to be able to meet this goal, we have developed a general framework for incorporating feedback from previous proof searches into the selection of the *given clause*. The corresponding learning theorem prover is organized as follows:

- In a first phase, a protocol of the inference steps is turned into a *proof derivation graph*. This graph is analyzed, and clauses contributing to the proof as well as clauses that are in certain sense *close* to the proof are selected as training examples.
- These clauses are then transformed into *annotated patterns*, where a pattern is a unique representation for a class of structurally identical clauses, and where an annotation describes the role of the clauses corresponding to the pattern in previous proof searches. The resulting sets of patterns are stored in a knowledge base and indexed by a feature vector describing the original proof problem.
- In the application phase, we use this feature vector to decide on a set of proof experiences to use to guide the new proof search. The corresponding patterns are recalled and fed into a learning algorithm. Newly generated clauses are evaluated using the resulting learned knowledge representation.

In the instance of the general framework described in this paper, the learning algorithm simply stores the clause patterns with their corresponding annotations. If a new clause matches a known pattern, its standard evaluation is modified based on the annotations.

Our implementation of the learning process is fully automatic. The only human intervention necessary is the a-priory selection of parameters for both the theorem prover and the analysis module that determines which clauses are used to represent a given proof search.

Fig. 1 shows a general sketch of this learning cycle as implemented in our prover E. We will discuss the different aspects of this cycle in the next sections.

3.1 Knowledge Acquisition

The first problem is to identify *which* search decisions from a given proof search should be used as input for a learning prover. As we wrote above, the starting point for our learning system is a complete listing of all inferences performed by the prover during a successful proof search. Such a listing basically consists of a number of steps, each of which contains a clause (the result of the step) and a justification for this clause. A justification is either *"initial"* for clauses from the problem specification, or a description of the inference generating this clause, including pointers to all clauses used in the premise of the inference.

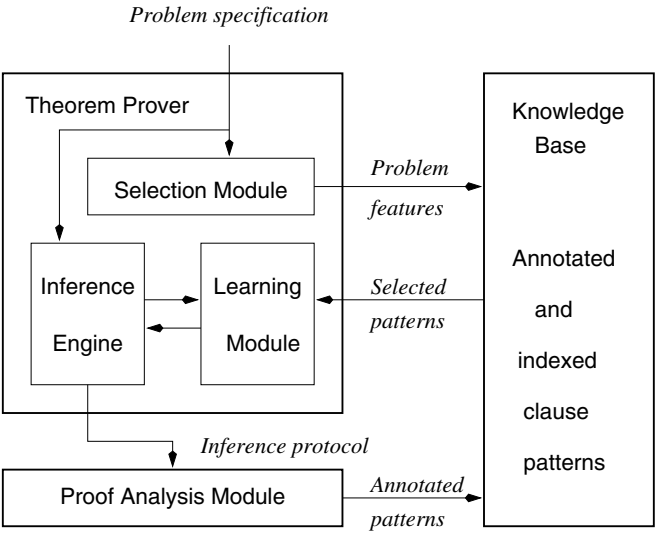


Fig. 1. Architecture of the learning theorem prover

An inference listing defines a directed *clause dependency graph*. In this graph, nodes correspond to the clauses (where different occurrences of the same clause are considered to be distinct objects and are represented by different nodes) and edges go from the premises of an inference to the conclusion. A path from an initial clause to the empty clause is called a *proof path*, clauses on a proof path are called *contributing* (to the proof) and all other clauses are called *superfluous*. The subgraph containing exactly the nodes and edges of proof paths is called the *proof object* or simply *proof*[3].

In a typical non-trivial proof search, superfluous clauses outnumber contributing clauses by a factor of at least 100. To keep the knowledge base manageable, and to balance the number of contributing and superfluous clauses we use as training examples, we select an appropriate number of superfluous clauses from those that are close to the proof, i.e. that can be derived in at most a small number of inferences from the contributing clauses. Additionally, we discard clauses that have never been selected as the *given clause*, as these clauses did not have a chance to contribute to a proof at all, and hence we have no information on the impact they would have had on the proof search.

[3] While proofs are often considered to be *trees*, generating provers reuse intermediate results, and thus each node (corresponding to a clause) can be referenced more than once, resulting in a general directed acyclic graph structure. The customary proof tree can be generated by recursively duplicating subgraphs referenced more than once, thus *unfolding* the graph. However, as this typically leads to an exponential explosion of the number of nodes, we consider the graph representation to be not only more natural, but also more suitable for our purposes.

All selected clauses are annotated with information about their role in the proof process. We found that the clause status (contributing or superfluous) and the distance of the clause from the closest proof path are useful for assigning evaluations to clauses.

3.2 Knowledge Representation

The next problem we have to tackle is to find an adequate representation of search decisions. This representation has to strike a fine line between generality and specificness. Equivalent search decisions should be represented by similar structures, but significantly different ones should still be distinct.

We achieve this compromise by the *generalization* of clauses into clause patterns. This serves two purposes. First, remember that clauses are multi-sets of literals, and are hence unordered. Thus, there are 2^n different ways to order the literals in a clause with n literals. Moreover, equational literals are again unordered pairs of terms, and can be arranged independently in each literal. Thus, any given clause with n literals has $n!\,2^n$ different syntactic representations. We want to have a unique representation for all these strictly equivalent but syntactically different clause representations.

Secondly, the transformation into patterns abstracts from the given signature of the problem. In our experience, specifications of problems from the same domain are often encoded using different function symbols. Moreover, there are analogous substructures even within a single specification. As an example, consider algebraic fields, which contain both an additive and a multiplicative group. We are therefore convinced that the loss of information resulting from the abstraction of signature information is more than compensated for by the ability to generalize to analogous situations. Earlier experiments for the unit-equational case [DS96,DS00] as well as the experimental results presented in this paper support this conviction.

We achieve these two goals by systematically replacing function symbols with new symbols from the set $\{f_{ij}|i,j \in \mathbf{N}\}$ (where the first index encodes arity, the second one distinguishes between different symbols of the same arity), and by reordering the terms and literals in the clause representation, until a representation is found that is minimal in a suitable ordering.

We use a modified lexicographic ordering that first compares terms based only on their size and structure (ignoring the symbols) and then compares terms lexicographically. The ordering is then lexicographically lifted to literal and clause representations. The search for the minimal clause representation is conducted using a conventional backtracking algorithm, which in the practical implementation is supported by a pre-ordering of terms and literals with the part of the ordering that is independent of function symbols and depends on the structure of terms only. The final result is then encoded as a first order term over an extended signature.

We use the reserved symbols *eq/2* and *neq/2* to encode positive and negative equations, respectively. The disjunction of the literals of a clause is encoded by using the reserved symbols *or/2* and *nil/0* as standard list constructors.

As an example, the clause

$$b = f(a, a) \vee a \neq b$$

is represented by the pattern

$$or(eq(f_{20}(f_{00}, f_{00}), f_{01}), or(neq(f_{00}, f_{01}), nil))$$

The knowledge base simply stores patterns of clauses with the annotations computed in the proof search analysis. Patterns are indexed by a vector of 15 feature values describing aspects of the original problem specification, and by arity frequency vectors for function symbols and predicate symbols. Features used include e.g. the number of unit, Horn and general clauses, the average size and depth of terms, and the average number of positive and negative literals per clause. The arity frequency vectors describe how many symbols of each arity are present in the problem signature. For a detailed description of the problem selection mechanism see [Sch00].

All relevant data is stored in the form of plain ASCII text files in a subdirectory on the hard disk. While it is possible to modify the knowledge base by hand, we have implemented tools to create a knowledge base and to automatically insert new or remove old proof examples.

3.3 Knowledge Application

The application phase is concerned with two main problems: The identification of suitable proof experiences and the application of the corresponding knowledge to the new proof search.

If a new proof problem is given to the learning theorem prover, the corresponding feature vector is computed. The prover then uses a standard distance measure (modified Euclidean distance) to determine a set of similar problems[4]. All clause patterns from selected proof experiences are stored into a pattern cache organized as a splay tree [ST85], a memory-efficient data structure suitable for very fast data retrieval. If the same pattern occurs more than once in the knowledge base, the corresponding annotations are combined. As a result, each of the patterns in the tree carries a tuple (p, d), where p describes the number of proofs the clauses corresponding to the pattern participated in, and d is the average distance of these clauses from the proof.

New clauses to be evaluated are transformed into patterns as well. If the resulting pattern is found in the cache, we use the annotation of the pattern to compute the evaluation. Otherwise, we use an annotation $(0, d_{max})$, where (d_{max}) is the largest distance in the cache, increased by one. In order to increase the stability of the prover and to help the system to fill gaps in the acquired knowledge, we combine a conventional heuristic with the learned knowledge.

[4] For pattern memorization, we get no benefit from selecting only similar problems, and hence always use all available training examples (see section 4 for a short discussion). However, if we use more sophisticated versions of *term space mapping* [Sch00], this selection becomes important.

Consider a clause C to be evaluated with an annotation of (p, d). A clause with a large value of p should get a better than standard evaluation (it has contributed to many proofs). A clause with a large value of d, on the other hand, usually has not contributed to proofs, and should receive a worse evaluation. Now assume that w is the weight assigned by a standard symbol counting evaluation function. The weight assigned by the learning heuristics is $lw(C) = w(1 + w_l norm\ (w_p\overline{p} + w_d\overline{d}))$, where $\overline{p}$ and $\overline{d}$ are normalized by dividing p and d by the largest corresponding values in any annotation in the pattern cache, where w_p and w_d determine the relative influence of these two variables, where *norm* rescales all evaluations for annotations in the cache to the interval $[0; 1]$, and where finally w_l determines the overall influence of the learning component. The double renormalization effect allows us to independently control the influence of the two parameters p and d, as well as the relative weight given to the conventional and the learned parts of the evaluation.

Please note that for positive values of w_l the resulting strategy is *fair* if the conventional base strategy is, i.e. it will never prefer an infinite number of clauses to any given clause.

4 Experimental Results

We have implemented the above concepts as a special case of our learning theorem prover E/TSM. To evaluate the approach, we have tested the resulting system on the set of all 3558 clause normal form (CNF) problems from the TPTP problem library [SSY94,SS97], version 2.3.0[5].

Clause normal form problems in the TPTP are either *unsatisfiable*, which implies that the underlying first-order problem does have a proof, or *satisfiable*, which means that a model exists that can serve as a counterexample for the original first-order problem. A problem can be *proved* by a prover if it can show the unsatisfiability. Alternatively, a prover can (in our case rarely) show that no proof is possible, i.e. that a *model* for the specification exists. We consider both of these case to be a *success* of the proof search. Termination of the proof search without a definitive answer about the state of the problem, either due to reaching a time limit or due to running out of some other limited resource, are considered failures.

To test the success of learning, we compare the learning heuristic to a standard symbol counting heuristic (which always picks the clause with the least number of symbols). Both heuristics are interleaved with first-in-first-out (always pick the oldest clause) using a pick-given ratio of 5 to 1. For the symbol counting heuristic, this combination corresponds to a widely used standard heuristic implemented e.g. in Otter [MW97]and Vampire [RV99]. E controlled

[5] The results presented in the following have been obtained in compliance with the guidelines for use of the TPTP. TPTP input files were unchanged except for removal of equality axioms and syntax transformation. All experiments were performed on a cluster of SUN Ultra 60 workstations running at 300 MHz. Memory was limited to 192 megabyte for each proof attempt.

by this conventional heuristic can prove 1602 out of the 3558 CNF problems in TPTP within a 300 second time limit on our hardware. It also finds 88 models, for total of 1690 successes.

The learning evaluation function uses the same symbol counting heuristic as a base and uses values of 5 for w_l, -20 for w_p and 20 for w_d (note that only the relative size of w_p and w_d are relevant due to the normalization). After preliminary experiments, we have always chosen *all* problems in the knowledge base as training examples. It turns out that any strong decrease in the number of problems used also impairs the performance, while the presence of more, probably very different problems does not hurt the pattern memorization based heuristic at all. This may be due to the fact that very different proof problems also generate syntactically different clause patterns, and hence do not influence the proof search at all.

In all cases, we have used a Knuth-Bendix ordering with constant weight for all symbols and a precedence that is induced by the symbol arities (symbols with a higher arity are bigger than symbols with a lower arity, order between symbols of the same arity is chosen at random). We also used a literal selection strategy that always selects the negative literal with the largest size difference between the two terms in non-positive clauses.

To demonstrate the ability of the learning heuristic to generalize to unseen problems, we have adapted *10 fold stratified cross validation*, a standard technique from the field of machine learning. The set of all proof problems is partitioned into 10 approximately equal sized sets (or *folds)*. Within each set, the number of problems that can be solved by the conventional heuristic is again approximately equal. We then use the proofs from nine out of the ten folds as training examples for the learning heuristic and apply it to the remaining fold. In particular, this means that none of the new proof problems is already known to the prover. The resulting knowledge bases each contain about 1450 training examples, with about 70000 clauses. These are mapped onto about 14000 distinct clause patterns. The knowledge bases have a size of about 5 megabyte for the functional part and of approximately 20 megabyte if we include the full, uncompressed set of training data for archival purposes.

Table 1 shows the results of the cross evaluation. We again used a time limit of 300 seconds for each proof attempt. As we can see, in each fold the learning heuristic performs better than the conventional heuristic. If we consider the set of all problems, this improvement is statistically significant. We can also see that all of the improvement is due to the finding of more proofs. Indeed, the learning heuristic does loose one model compared to the base heuristic.

If we consider the performance of the 10 learning heuristics as a whole, we find that there are 19 problems that are only proved by the conventional heuristic, but not by the learning heuristic used on the given problem. On the other hand, there are 138 problems that are solved by the appropriate learning heuristic, but not by the conventional one. Thus, while the learning heuristics do not prove a strict superset of the problems solved by the conventional one, they can solve a fairly large number of new, hard problems.

Table 1. Results of 10-fold stratified cross validation

Fold	Conventional			Learning		
	Proofs	Models	Successes	Proofs	Models	Successes
1	161	12	173	180	12	192
2	161	7	168	172	7	179
3	160	12	172	167	12	179
4	160	10	170	171	10	181
5	160	6	166	173	6	179
6	160	14	174	168	13	181
7	160	6	166	175	6	181
8	160	5	165	165	5	170
9	160	10	170	179	10	189
10	160	6	166	171	6	177
Average	160.2±0.422	8.8±3.190	169.0±3.266	172.1±4.886	8.7±3.020	180.8±6.088
Total	1602	88	1690	1721	87	1808

We can also compare the different heuristics on subsets of the TPTP: Problems containing unit clauses only, non-unit Horn problems, and problems with general clauses. Each class is once more split according to the presence of an equational sub-theory. Table 2 shows the results.

Again we can see that the learning heuristic is superior nearly across the board. However, the gain is most significant for the case of non-horn-problems with equality (which also is generally considered to be the hardest class of problems in the TPTP), and is fairly small for unit problems. This may be correlated with the size of the corresponding class. The improvements achieved by the learning heuristic increase with the size of the problem class, which supports the plausible idea that more suitable training examples lead to better performance on unknown problems.

Table 2. Number of successes by problem class

Problem class	Size	Conventional	Learning	Successes gained
Unit, no equality	11	11	11	0.00%
Unit with equality	447	289	292	1.04%
Horn, no equality	609	464	492	6.03%
Horn with equality	507	291	308	5.84%
General, no equality	766	307	333	8.47%
General with equality	1218	328	372	13.41%
Total	3558	1690	1808	6.98%

Finally, we can compare both heuristics on the TPTP domains. A TPTP domain is a set of problems dealing with similar structures. However, TPTP

domains are usually more diverse than typical application domains, because TPTP domains have been created *a posteriori* to organize a large number of existing proof problems, and do not result from a systematic axiomatization of a given application area.

Table 3. Number of successes by TPTP domain

Problem class	Size	Conventional	Learning	Successes gained
ALG	10	2	2	0.00%
ANA	19	2	2	0.00%
BOO	68	55	60	9.09%
CAT	58	54	53	-1.85%
CID	4	1	1	0.00%
CIV	14	11	11	0.00%
COL	160	94	94	0.00%
COM	6	5	5	0.00%
FLD	281	24	53	120.83%
GEO	165	98	101	3.06%
GRA	1	1	1	0.00%
GRP	376	254	251	-1.18%
HEN	64	62	63	1.61%
KRS	17	17	17	0.00%
LAT	35	12	16	33.33%
LCL	503	262	296	12.98%
LDA	23	19	19	0.00%
MGT	0	0	0	–
MSC	13	10	9	-10.00%
NUM	309	33	37	12.12%
PLA	30	5	5	0.00%
PRV	9	8	8	0.00%
PUZ	60	53	53	0.00%
RNG	98	41	45	9.76%
ROB	36	14	14	0.00%
SET	695	171	212	23.98%
SYN	480	377	376	-0.27%
TOP	24	5	4	-20.00%
Total	3558	1690	1808	6.98%

Table 3 shows the results by domain. Now we see a more diverse picture. In many domains, especially smaller ones, learning does not improve the performance of the prover much, or even leads to slightly worse performance (although this is typically not significant). Most of the gain comes from a couple of larger domains. FLD, LAT and SET are particularly impressive. Again, we get the impression that an increased number of suitable training examples improves the performance of the learning heuristic. This makes it likely that the learning

heuristics are particularly useful in domains like verification or software reuse, where a large number of similar problems have to be solved over and over again, possibly with slight modifications.

The successes of the learning heuristics are achieved despite a significant computational overhead. It takes about 15 seconds to read the knowledge base into memory, determine the set of similar problems, and to prepare the pattern cache with pre-computed evaluations. Moreover, the inference rate is about 20% lower as for the less complex conventional strategies. Profiling shows that most of this overhead is spent on the transformation of new clauses into patterns. As this operation depends only on the single clause, and not on the total size of the clause database (as e.g. superposition and subsumption), this cost is relatively less significant for large problems, and will likely become less important as advances in hardware allow us to deal with even bigger search spaces.

5 Conclusion

In this paper, we have described a fully automatic learning theorem prover based on the memorization of clause patterns from previous proof searches. The learning heuristics were able to significantly outperform a conventional base heuristic, especially on hard problems. Our results show that clause pattern memorization is an adequate method for learning search control knowledge in domains where enough previous proof experiences are available. We find that time spent on good search guidance is usually well spent. While our results have been obtained with a superposition-based prover, there is no reason to suppose that it would not carry over to other saturating systems, including e.g. inductive provers.

In the future, we will combine the basic learning cycle and the abstraction provided by clause patterns with other learning algorithms. We have already investigated various versions of *term space mapping* (strictly speaking, pattern memorization is subsumed as a special case) with good success [Sch00]. We may also use *folding architecture networks* [GK96,SKG97], a much more powerful, but rather slow learning algorithm based on the neural network paradigm.

We also will try to combine proof experiences generated using a wide variety of different search heuristics, and to work with much larger knowledge bases. In all of our experiments to date, the performance of heuristics based on pattern memorization has increases significantly with an increase in stored proof experiences. We therefore expect significant synergy effects from this approach.

Finally, our prototypical implementation can significantly benefit from a clean-up and partial redesign of the time-critical sections of the code.

References

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[BG98] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (1) of *Applied Logic Series*, chapter 11, pages 353–397. Kluwer Academic Publishers, 1998.

[BRLP98] J. Van Baalen, P. Robinson, M. Lowry, and T. Pressburger. Explaining Synthesized Software. In *Proc. of the 13th IEEE Conference on Automated Software Engineering, Honolulu*. IEEE, 1998.

[DFGS99] J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from Previous Proof Experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999. (also to be published as a SEKI report).

[DGHW97] B.I. Dahn, J. Gehne, T. Honigmann, and A. Wolf. Integration of Automated and Interactive Theorem Proving in ILF. In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 57–60. Springer, 1997.

[DS96] J. Denzinger and S. Schulz. Learning Domain Knowledge to Improve Theorem Proving. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 62–76. Springer, 1996.

[DS00] J. Denzinger and S. Schulz. Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts. *Journal of Information and Computation*, 162:59–79, 2000.

[DW97] B.I. Dahn and C. Wernhard. First Order Proof Problems Extracted from an Article in the MIZAR Mathematical Library. In *Proceedings of the 1st FTP, Linz*, pages 58–62. RISC Linz, Austria, 1997.

[FS97] B. Fischer and J. Schumann. SETHEO Goes Software-Engineering: Application of ATP to Software Reuse. In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 65–68. Springer, 1997.

[Fuc96] M. Fuchs. Experiments in the Heuristic Use of Past Proof Experience. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 523–537. Springer, 1996.

[Fuc97] M. Fuchs. *Learning Search Heuristics for Automated Deduction*. Number 34 in Forschungsergebnisse zur Informatik. Verlag Dr. Kovač, 1997. Accepted as a Ph.D. Thesis at the Fachbereich Informatik, Universität Kaiserslautern.

[GK96] C. Goller and A. Küchler. Learning Task-Dependent Distributed Representations by Backpropagation Through Structure. In *Proc. of the ICNN-96*, volume 1, pages 347–352. IEEE, 1996.

[GP00] S. Gürdens and René Peralta. Validation of Cryptographic Protocols by Efficient Automatic Testing. In J. Etheredge and B. Manaris, editors, *Proc. of the 13th FLAIRS, Orlando*, pages 7–12. AAAI Press, 2000.

[HLS+96] D. Hutter, B. Langenstein, C. Sengler, J.H. Siekmann, W. Stephan, and A. Wolpers. Verication support environment (VSE). *Journal of High Integrity Systems*, 1(6):523–531, 1996.

[McC97] W.W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 3(19):263–276, 1997.

[MW97] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[RSS95] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Proc. of the 10th Annual Conference on Computer Assurance, Gaithersburg*. IEEE Press, 1995.

[RV99] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 292–2296. Springer, 1999.

[Sch97] J. Schumann. Automatic Verification of Cryptographic Protocols with SETHEO. In W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 87–11. Springer, 1997.

[Sch99] S. Schulz. System Abstract: E 0.3. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 297–391. Springer, 1999.

[Sch00] S. Schulz. *Learning Search Control Knowledge for Equational Deduction*. Number 230 in DISKI. Akademische Verlagsgesellschaft Aka GmbH Berlin, 2000. Ph.D. Thesis, Fakultät für Informatik, Technische Universität München.

[Sch01] S. Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, number 2083 in LNAI, pages 370–375. Springer, 2001.

[SKG97] S. Schulz, A. Küchler, and C. Goller. Some Experiments on the Applicability of Folding Architecture Networks to Guide Theorem Proving. In D.D. Dankel II, editor, *Proc. of the 10th FLAIRS, Daytona Beach*, pages 377–381. Florida AI Research Society, 1997.

[SS97] C.B. Suttner and G. Sutcliffe. The TPTP Problem Library (TPTP v2.1.0). Technical Report AR-97-01 (TUM), 97/04 (JCU), Institut für Informatik, Technische Universität München, Munich, Germany/Department of Computer Science, James Cook University, Townsville, Australia, 1997. Jointly published.

[SSY94] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In A. Bundy, editor, *Proc. of the 12th CADE, Nancy*, number 814 in LNAI, pages 252–266. Springer, 1994.

[ST85] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[SWL+94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In A. Bundy, editor, *Proc. of the 12th CADE, Nancy*, number 814 in LNAI, pages 341–355. Springer, 1994.

[WAB+99] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, G. Jung, E. Keen, C. Theobalt, and D. Topic. System Abstract: SPASS Version 1.0.0. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 378–382. Springer, 1999.

[Wei99] C. Weidenbach. Toward an Automatic Analysis of Security Protocols in First-Order Logic. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 314–328. Springer, 1999.

The most recent version of the E equational theorem prover and the other programs used for the experiments described in this paper are available from `http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`.

Intelligent Structuring and Reducing of Association Rules with Formal Concept Analysis

Gerd Stumme[1], Rafik Taouil[2], Yves Bastide[3], Nicolas Pasquier[4], and Lotfi Lakhal[5]

[1] Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB), Universität Karlsruhe (TH), D–76128 Karlsruhe, Germany; stumme@aifb.uni-karlsruhe.de

[2] INRIA Lorraine, LORIA, BP 239, F–54506 Vandœuvre–lès–Nancy, France; rafik.taouil@loria.fr

[3] Laboratoire d'Informatique (LIMOS), Université Blaise Pascal, Complexe Scientifique des Cézeaux, 24 Av. des Landais, F–63177 Aubière Cedex, France; bastide@libd2.univ-bpclermont.fr

[4] I3S, CNRS UMR–6070, Université de Nice – Sophia Antipolis, Les Algorithmes, 2000 route des Lucioles, F–06903 Sophia Antipolis, France; nicolas.pasquier@unice.fr

[5] LIM, CNRS FRE-2246, Université de la Méditerranée, Case 90, 163 Avenue de Luminy, F–13288 Marseille Cedex 9, France; lotfi.lakhal@lim.univ-mrs.fr

Abstract. Association rules are used to investigate large databases. The analyst is usually confronted with large lists of such rules and has to find the most relevant ones for his purpose. Based on results about knowledge representation within the theoretical framework of Formal Concept Analysis, we present relatively small bases for association rules from which all rules can be deduced. We also provide algorithms for their calculation.[1]

1 Introduction and Motivation

One of the core tasks of *Knowledge Discovery in Databases* (*KDD*) is the mining of association rules (conditional implications). *Association rules* are statements of the type '67 % of the customers buying cereals and sugar also buy milk (where 7% of all customers buy all three items)'. The task of mining association rules is to determine all rules whose *confidences* (67 % in the example) and *supports* (7 % in the example) are above user-defined thresholds. Since the problem was stated [1], various approaches have been proposed for an increased efficiency of rule discovery in very large databases [2,7,11,30,31]. However, fully taking advantage of exhibited rules means providing capabilities to handle them. The problem is especially critical when collected data is highly correlated or dense, like in statistical databases [11]. For instance, when applied to a census dataset of 10,000

[1] This paper is a revised and extended version of a presentation given at the workshop "Bases de Données Avancées", Bordeaux, France, 1999 [29], and of the technical report [37].

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 335–350, 2001.

objects, each of which characterized by values of 73 attributes, experiments result in more then 2,000,000 rules with support and confidence greater than or equal 90%. Thus the question arises: How can long lists of association rules be reduced in size?

Approaches addressing the described issue provide users with mechanisms for filtering rules, for instance by user defined templates [4,21], Boolean [26,35] or SQL-like [25] operators or by introducing further measures of "usefulness" [8]; or they attempt to minimize the number of extracted rules a priori by using information about taxonomies [17,19,34] or by applying statistical measures like Pearson's correlation or the χ^2-test [10]. All these approaches have in common that they lose some information.

Our approach, on the other hand, allows us to significantly reduce the number of rules without losing any information. We extract only a subset of all association rules, called *basis*, from which all other rules can be derived. This approach is orthogonal to the ones mentioned above and can be combined with them.

We make use of techniques of *Formal Concept Analysis* (FCA). Formal Concept Analysis [41,15] arose as a mathematical theory for the formalization of the concept of 'concept' in the early 80ies and is nowadays considered as an AI theory. It has since then grown to a technique for data analysis, information retrieval, and knowledge representation with over 200 applications, for analyzing flight movements at Frankfurt Airport [20], for studying semantics of German speech-act verbs [16], for examining the medical nomenclature system SNOMED [33], for IT-security management [9], and for database marketing [18]. FCA provides a framework for KDD, especially for conceptual clustering and association rules. A broad discussion of the role of Formal Concept Analysis in data analysis, decision support, and KDD is provided in [18] and [36].

We use results of Duquenne and Guigues ([12], cf. also [15]) and Luxenburger [22,23]. The former have studied bases (i. e., minimal non-redundant sets of rules from which all other rules can be derived) for association rules with 100 % confidence, and the latter association rules with less than 100 % confidence, but neither of them considered the support of the rules. We adopt their results to association rules (where both the support and the confidence are considered) and provide algorithms for computing the new bases by using *iceberg concept lattices* [39]. We follow an approach in two steps. In the first step, we compute the iceberg concept lattice for the given parameters. It consists of all FCA concepts whose extents exceed the user-defined minimum support. In the second step, we derive the bases for the association rules. In this paper, we focus on the second step. For the first step, we refer to the Pascal [6] and Titanic [38] algorithms.

This two-step approach has two advantages compared to the classical two-step approach [2] (which computes all frequent itemsets as intermediate result, and not only those which are intents of frequent FCA concepts):

1. It allows to determine bases for non-redundant association rules and thus to prune redundancy.

2. It speeds up the computation, especially for strongly correlated data or when the minimum support is low.

In [5], we have presented another pair of bases, which provide rules with minimal antecedents and maximal consequents. Compared to the results presented here, they have the disadvantage of a higher total number of rules. For the approximate rules, M. Zaki has presented similar results in [44]. However, he does not provide inference rules for support and confidence derivation, does not discuss minimality of his results, and does not provide algorithms for the computation of the bases.

The remainder of this paper is as follows. After having recalled some basic definitions in Section 2, we introduce two bases for association rules in Section 3: the *Duquenne-Guigues basis for exact association rules* (i. e., for all rules with a 100% confidence), and the *Luxenburger basis for approximate association rules* (i. e., with a confidence $< 100\%$). In Section 4, algorithms are given which compute the two bases. We conclude the paper with the presentation of experimental results (Section 5) and a discussion of future work (Section 6).

2 Formal Concept Analysis and the Association Rule Framework

In this section, we briefly recall the basic notions of Formal Concept Analysis [41,15] and the association rule problem [1]. For a more extensive introduction into Formal Concept Analysis refer to [15].

Definition 1. *A* formal context *is a triple* $\mathbb{K} := (G, M, R)$ *where* G *and* M *are sets and* $R \subseteq G \times M$ *is a binary relation. A* data mining context *(or* dataset*) is a formal context where* G *and* M *are finite sets. Its elements are called* objects *and* items*, respectively.* $(o, i) \in R$ *is read as "object* o *is related to item* i*".*

For $O \subseteq G$*, we define* $f(O) := \{i \in M \mid \forall o \in O\colon (o, i) \in R\}$*; and for* $I \subseteq M$*, we define dually* $g(I) := \{o \in G \mid \forall i \in I\colon (o, i) \in R\}$*. A* formal concept *is a pair* $(O, I) \in \mathfrak{P}(G) \times \mathfrak{P}(M)$ *with* $f(O) = I$ *and* $g(I) = O$*. O is called* extent *and* I *is called* intent *of the concept. The set of all concepts of a formal context* $\mathbb{K}$ *together with the partial order* $(O_1, I_1) \leq (O_2, I_2) :\Longleftrightarrow O_1 \subseteq O_2$ $(\Longleftrightarrow I_2 \subseteq I_1)$ *is a complete lattice, called* concept lattice *of* $\mathbb{K}$*.*

In this setting, we call each subset of M *also* itemset*, and each intent* I *also* closed itemset *(since it satisfies the equation* $I = f(g(I))$*). For two closed itemsets* I_1 *and* I_2*, we note* $I_1 \prec I_2$ *if* $I_1 \subset I_2$ *and if there does not exist a closed itemset* I_3 *with* $I_1 \subset I_3 \subset I_2$*.*[2]

In the following, we will use the composed function $h := f \circ g\colon \mathfrak{P}(M) \to \mathfrak{P}(M)$ which is a closure operator on M (i. e., it is extensive, monotonous, and idempotent). The related closure system (i. e., the set of all $I \subseteq M$ with $h(I) = I$) is exactly the set of the intents of all concepts of the context.

[2] We write $X \subset Y$ if and only if $X \subseteq Y$ and $X \neq Y$.

Definition 2. *Let $I \subseteq M$, and let minsupp, minconf* $\in [0,1]$. *The* support count *of the itemset I in $\mathbb{K}$ is $supp(I) := \frac{|g(I)|}{|G|}$. I is said to be* frequent *if $supp(I) \geq minsupp$. The set of all frequent itemsets of a context is denoted FI.*

An association rule *is a pair of itemsets I_1 and I_2, denoted $I_1 \to I_2$, where $I_2 \neq \emptyset$. I_1 and I_2 are called* antecedent *and* consequent *of the rule, respectively. The* support *and* confidence *of an association rule $r := I_1 \to I_2$ are defined as follows: $supp(r) := \frac{|g(I_1 \cup I_2)|}{|G|}$, $conf(r) := \frac{supp(I_1 \cup I_2)}{supp(I_1)}$. If $conf(r)=1$, then r is called* exact association rule *(or* implication*), otherwise r is called* approximate association rule.

An association rule r holds *in the context if $supp(r) \geq minsupp$ and $conf(r) \geq minconf$. The set of all association rules holding in $\mathbb{K}$ for given minsupp and minconf is denoted AR.*

Remark 1. The definition of association rules often includes the additional condition $I_1 \cap I_2 = \emptyset$. This condition helps pruning rules which are obviously redundant, as $I_1 \to I_2$ and $I_1 \to I_2 \setminus I_1$ have same support and same confidence. In this paper, we omit the condition, in order to simplify definitions. When discussing the algorithms, however, we will use the condition since it saves memory.

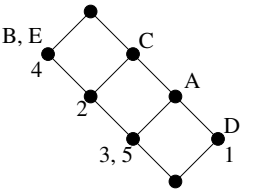
The association rule framework has first been formulated in terms of Formal Concept Analysis independently in [28], [37], and [42]. [28] provided also the first algorithm (named Close) based on this approach.

Example 1. An example data mining context $\mathbb{K}$ consisting of five objects (identified by their OID) and five items is given in Figure 1 together with its concept lattice. The association rules holding for $minsupp = 0.4$ and $minconf = 1/2$ are shown in the lower table.

In the *line diagram*, the name of an object g is always attached to the node representing the smallest concept with g in its extent; dually, the name of an attribute m is always attached to the node representing the largest concept with m in its intent. This allows us to read the context relation from the diagram because an object g has an attribute m if and only if there is an ascending path from the node labeled by g to the node labeled by m. The extent of a concept consists of all objects whose labels are below in the diagram, and the intent consists of all attributes attached to concepts above in the hierarchy. For example, the concept labeled by 'A' has $\{1,3,5\}$ as extent, and $\{\mathrm{A},\mathrm{C}\}$ as intent.

An example for an exact rule (implication) which holds in the context is $\{\mathrm{A},\mathrm{B}\} \to \{\mathrm{C},\mathrm{E}\}$. It can also be read directly in the line diagram: the largest concept having both A and B in its intent is the one labeled by 3 and 5, and it is below or equal to (here the latter is the case) the largest concept having both C and E in its intent. This implication can be derived from two simpler implications, namely $\{\mathrm{A}\} \to \{\mathrm{C}\}$ and $\{\mathrm{B}\} \to \{\mathrm{E}\}$. The aim of the frequent Duquenne-Guigues-basis which we introduce in the next section is to provide only a minimal, non-redundant set of implications to the user. That basis will include the two simpler implications.

	A	B	C	D	E
1	×		×	×	
2		×	×		×
3	×	×	×		×
4		×			×
5	×	×	×		×



Exact rule	Supp	Approximate rule	Supp	Conf	Approximate rule	Supp	Conf	Approximate rule	Supp	Conf
ABC → E	0.4	BCE → A	0.4	2/3	C → ABE	0.4	2/4	C → BE	0.4	3/4
ABE → C	0.4	AC → BE	0.4	2/3	E → ABC	0.4	2/4	E → BC	0.6	3/4
ACE → B	0.4	BE → AC	0.4	2/4	A → BC	0.4	2/3	A → B	0.4	2/3
AB → CE	0.4	CE → AB	0.4	2/3	B → AC	0.4	2/4	B → A	0.4	2/4
AE → BC	0.4	AC → B	0.4	2/3	C → AB	0.4	2/4	C → A	0.6	3/4
AB → C	0.4	BC → A	0.4	2/3	A → BE	0.4	2/3	A → E	0.4	2/3
AB → E	0.4	BE → A	0.4	2/4	B → AE	0.4	2/4	E → A	0.4	2/4
AE → B	0.4	AC → E	0.4	2/3	E → AB	0.4	2/4	C → B	0.6	3/4
AE → C	0.4	CE → A	0.4	2/3	A → CE	0.4	2/3	C → E	0.6	3/4
BC → E	0.6	BE → C	0.6	3/4	C → AE	0.4	2/4	E → C	0.6	3/4
CE → B	0.6	A → BCE	0.4	2/3	E → AC	0.4	2/4	∅ → E	0.8	4/5
A → C	0.6	B → ACE	0.4	2/4	B → CE	0.6	3/4	∅ → BE	0.8	4/5
B → E	0.8	∅ → C	0.8	4/5	∅ → AC	0.6	3/5	∅ → BC	0.6	3/5
E → B	0.8	∅ → A	0.6	3/5	∅ → B	0.8	4/5	∅ → CE	0.6	3/5
								∅ → BCE	0.6	3/5

Fig. 1. The example data mining context $\mathbb{K}$ and its concept lattice. The table shows all association rules that hold in $\mathbb{K}$ for $minsupp = 0.4$ and $minconf = 1/2$.

At the end of this section, we give some simple facts about association rules. We will refer to them later as derivation rules.

Lemma 1. *Rules 1 and 2 hold for* $\phi \in \{\mathrm{conf}, \mathrm{supp}\}$.

1. $\phi(X \to Y) = \phi(X \to Y \setminus Z)$, *for all* $Z \subseteq X \subseteq M$, $Y \subseteq M$.
2. $\phi(h(X) \to h(Y)) = \phi(X \to Y)$, *for all* $X, Y \subseteq M$.
3. $\mathrm{conf}(X \to Y) = p \wedge \mathrm{conf}(Y \to Z) = q \implies \mathrm{conf}(X \to Z) = p \cdot q$, *for all frequent concept intents* $X \subset Y \subset Z$.

3'. $\mathrm{supp}(X \to Z) = \mathrm{supp}(Y \to Z)$, *for all* $X, Y \subseteq Z$.

4. $\mathrm{conf}(X \to X) = 1$, *for all* $X \subseteq M$.

Proof. The proofs for the confidence are given in [23].

1. $\mathrm{supp}(X \to Y) = \mathrm{supp}(X \to Y \setminus Z)$ follows from $X \cup Y = X \cup (Y \setminus Z)$ and the definition of the support count.
2. $\mathrm{supp}(h(X) \to h(Y)) = \mathrm{supp}(X \to Y)$ follows from $g(h(X) \cup h(Y)) = g(h(X)) \cap g(h(Y)) = g(f(g(X))) \cap g(f(g(Y))) = g(X) \cap g(Y) = g(X \cup Y)$ by using the facts $g(f(g(X))) = g(X)$ and $g(X \cup Y) = g(X) \cap g(Y)$ provided in [15].

3'. $\mathrm{supp}(X \to Z) = \frac{|g(X \cup Z)|}{|G|} = \frac{|g(Z)|}{|G|} = \frac{|g(Y \cup Z)|}{|G|} = \mathrm{supp}(Y \to Z)$ □

Frequent closed itemset	support
∅	1.0
{C}	0.8
{AC}	0.6
{BE}	0.8
{BCE}	0.6
{ABCE}	0.4

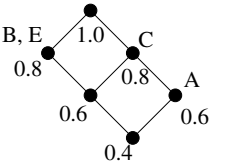


Fig. 2. Frequent closed itemsets extracted from $\mathbb{K}$ for *minsupp* = 0.4.

3 Bases for Association Rules

In this section, we recall the definition of *iceberg concept lattices* and show that one can derive all frequent itemsets and association rules from them. Then we characterize the *Duquenne-Guigues basis for exact association rules* and the *Luxenburger basis for approximate association rules* and show that all other association rules can be derived from these two bases.

Definition 3. *A concept* (O, I) *is called* frequent concept *if* $supp(I)\,(= \frac{|O|}{|G|}) \geq$ *minsupp. The set of all frequent concepts is called* iceberg concept lattice*. An itemset* I *is called* frequent intent *(or* frequent closed itemset*) if it is intent of a frequent concept (i. e., its support is at least minsupp). The set of all frequent closed itemsets in* $\mathbb{K}$ *is denoted* FC*.*

Example 2. The frequent closed itemsets in the context $\mathbb{K}$ for *minsupp*=0.4 are presented in Figure 2 together with the semi-lattice of all frequent concepts. Both the table and the diagram provide the same information. Note that, in general, the set of frequent concepts is not a lattice, but only a semi-lattice (consider e. g. *minsupp*= 0.5 in the example).

Lemma 2 ([31]). *i) The support of an itemset* I *is equal to the support of the smallest closed itemset containing* I*, i. e.,* $supp(I) = supp(h(I))$*.*

ii) The set of maximal frequent itemsets $\{I \in FI \mid \nexists I' \in FI\colon I \subset I'\}$ *is identical to the set of maximal frequent closed itemsets* $\{I \in FC \mid \nexists I' \in FC\colon I \subset I'\}$*.*

The next theorem shows that the set of frequent closed itemsets with their support is a small collection of frequent itemsets from which all frequent itemsets with their support and all association rules can be derived. I. e., it is a condensed representation in the sense of Mannila and Toivonen [24]. This theorem follows from Lemma 2.

Theorem 1. *All frequent itemsets and their support, as well as all association rules holding in the dataset, their support, and their confidence can be derived from the set* FC *of frequent closed itemsets with their support.*

3.1 Duquenne-Guigues Basis for Exact Association Rules

Next we present the Duquenne-Guigues basis for exact association rules. It is based on the following closure operator.

Theorem 2. *The set $FI \cup \{M\}$ is a closure system on M, and its related closure operator $\overline{\cdot}$ is given by $\overline{I} := h(I)$ if $supp(I) \geq minsupp$ and $\overline{I} := M$ else.*

Proof. The set of all frequent itemsets together with M is a closure system, as well as the set of all concept intents. Hence $FI \cup \{M\}$ is, as intersection of those two closure systems, also a closure system. The proof of the fact that $\overline{\cdot}$ is the corresponding closure operator is straightforward. □

Our basis adopts the results of [12] to the association rule framework, where additionally the support of the rules has to be considered.

Definition 4. *An itemset $I \subseteq M$ in $\mathbb{K}$ is a* $\overline{\cdot}$–pseudo-closed itemset *(or* pseudo-closed itemset *for short)* [3] *if $\overline{I} \neq I$ and for all pseudo-closed itemsets J with $J \subset I$, we have $\overline{J} \subset I$. The set of all frequent pseudo-closed itemsets in $\mathbb{K}$ is denoted FP, the set of all infrequent pseudo-closed itemsets is denoted IP. In the (unlikely) case that all itemsets are frequent except the whole set M, we let $IP := \{M\}$ (in order to distinguish this situation from the one where all itemsets are frequent).*

The Duquenne-Guigues basis for exact association rules *(or* frequent Duquenne-Guigues basis*) is defined as the tuple $FDG := (\mathcal{L}, IP)$ with $\mathcal{L} := \{I_1 \to h(I_1) \mid I_1 \in FP\}$ and IP as defined above.*

Theorem 3. *From the Duquenne-Guigues basis for exact association rules one can derive all exact association rules holding in the dataset by applying the following rules. Rules ii) to iv) can be applied to $\mathcal{L}$ as long as they do not contradict (i).*

i) If there exists $I \in IP$ with $I \subseteq I_1 \cup I_2$, then $I_1 \to I_2$ does not hold (because its support is too low).
ii) $X \to X$ holds.
iii) If $X \to Z$ holds, then also $X \cup Y \to Z$.
iv) If $X \to Y$ and $Y \cup Z \to W$ hold, then also $X \cup Z \to W$.

Proof. We only sketch the proof here, which applies results of [12] (see also [15]). One has to check that $\mathcal{L} \cup \{I \to M \mid I \in IP\}$ is the Duquenne-Guigues-basis (in the traditional sense, cf. to [12,15]) of the closure system $FC \cup \{M\}$. Rule (i) reflects the implications of the form $I \to M$. □

The Duquenne-Guigues basis for exact association rules is not only minimal with respect to set inclusion, but also minimal with respect to the number of rules in $\mathcal{L}$ plus the number of elements in IP, since there can be no complete set with

[3] We do not consider pseudo-closed itemsets with respect to other closure operators than $\overline{\cdot}$ (especially not with respect to h) in this paper.

fewer rules than there are frequent pseudo-closed itemsets [12,15]. Observe that, although it is possible to derive all exact association rules from the Duquenne-Guigues basis, it is not possible in general to determine their support.[4]

Example 3. The set of frequent pseudo-closed itemsets of $\mathbb{K}$ for *minsupp*=0.4 and *minconf*=1/2 is $FP = \{\{A\}, \{B\}, \{E\}\}$, the set of infrequent pseudo-closed itemsets is $IP = \{\{D\}\}$. The Duquenne-Guigues basis is presented in Figure 3.

3.2 Luxenburger Basis for Approximate Association Rules

In [22,23], M. Luxenburger discusses bases for partial implications. A *partial implication* is an association rule where the support is not considered. He observed that it is sufficient to consider rules between concept intents only, since $\text{conf}(X \to Y) = \text{conf}(h(X) \to h(Y))$. However, his derivation process does not only consist of deduction rules which can be applied in a straightforward manner, but it requires to solve a system of linear equations.

In the KDD process, however, we have to consider the trade-off between the amount of information presented to the user, and the degree of its explicitness. The appearance of the system of linear equations indicates that Luxenburger's results are in favor for a minimal amount of information presented, and against a higher degree of explicitness. As one of the requirements to KDD is that the results should be "ultimately understandable" [13], we want to emphasize more on the explicitness of the results. Therefore we restrict now the expressiveness of the derivation process. This forces the association rules presented to the user to be more explicit.[5]

In the sequel, we consider the derivation rules given in Lemma 1. We present a basis for the approximate association rules for these derivation rules.

Definition 5. *The* Luxenburger basis for approximate association rules *is given by* $LB := \{(r, supp(r), conf(r)) \mid r = I_1 \to I_2,\ I_1, I_2 \in FC,\ I_1 \prec I_2,\ conf(r) \geq minconf,\ supp(I_2) \geq minsupp\}$.

Theorem 4. *From the Luxenburger basis LB for approximate association rules one can derive all association rules holding in the dataset together with their support and their confidence by using the rules given in Lemma 1. Furthermore, LB is minimal (with respect to set inclusion) with this property.*

Proof. In order to determine if an association rule $r := I \to J$ holds in a context (and for determining its support and its confidence) one can consider the rule $I' \to J'$ with $I' := h(I)$ and $J' := h(I \cup J)$ which has (by Rules 1 & 2) the same support and the same confidence. If $I' = J'$, then $conf(r) = 1$ and $supp(r) = \text{supp}(I')$. If $I' \neq J'$, then exists a path of approximate rules,

[4] Even if the support for all rules in the basis is known. With the knowledge about all frequent closed itemsets and their support however, this is possible (see Theorem 1).

[5] Note that in the KDD setting the user will never actually perform longer series of inference steps.

Luxenburger basis		
Approximate rule	Support	Confidence
BCE → A	0.4	2/3
AC → BE	0.4	2/3
BE → C	0.6	3/4
C → BE	0.6	3/4
C → A	0.6	3/4
∅ → BE	0.8	4/5
∅ → C	0.8	4/5

Duquenne-Guigues basis	
$\mathcal{L}$	(Support)
A → C	0.6
B → E	0.8
E → B	0.8
$IP = \{\{D\}\}$	

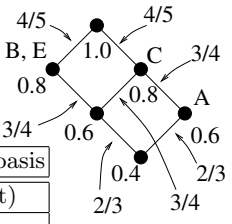


Fig. 3. Duquenne-Guigues and Luxenburger bases for *minsupp*=0.4 and *minconf*=1/2.

i. e., there are frequent closed itemsets $I_1, \ldots, I_n$ with $I_i \rightarrow I_{i+1} \in LB$ and $I' = I_1$ and $I_n = J'$. Support and confidence of r can now be determined by $supp(r) = supp(I_n)$ (Rule 3') and $conf(r) = \Pi_{i=1}^{n-1} conf(I_i \rightarrow I_{i+1})$ (Rule 3).

Now we show the minimality of LB. Let $r := I \rightarrow J \in LB$. We show that the confidence of r cannot be derived from $LB \setminus \{r\}$ by applying the rules of Lemma 2. Rule 1 cannot be applied forward since J already contains I. It cannot be applied backward because of $I \prec J$. Rule 2 cannot be applied forward since $I = h(I)$ and $J = h(J)$. It cannot be applied backward as LB contains only rules with closed antecedent and closed consequent. Rule 3 cannot be applied since there is no $K \subset M$ with $I \rightarrow K \in LB \setminus \{r\}$ and $K \rightarrow J \in LB \setminus \{r\}$ (because of $I \prec J$). Rule 4 cannot be applied since $I \not= J$. □

Remark 2. A basis in the sense of [23] is a maximal spanning tree of our basis (when considered as *undirected* graph) containing at most one rule with M as conclusion.[6]

Example 4. The Luxenburger basis for approximate association rules of $\mathbb{K}$ for *minsupp*=0.4 and *minconf*=1/2 is also presented in Figure 3. It provides the same information as the list in Figure 1, but in a more condensed form. The Luxenburger basis is visualized in the line diagram in Figure 3: From its definition it is clear, that each approximate rule in the basis corresponds to (at most)[7] one edge in the diagram. The edge is labeled by the confidence of the rule (as a fraction), and its lower vertice is labeled by its support (as a rational). Implications (exact rules) can be read in the diagram in the standard way described in Section 2.

As example for the proof of Theorem 4, let us check if $\{B\} \rightarrow \{A\}$ holds in the context for *minsupp*=0.4 and *minconf*=1/2. We have $I := \{B\}$ and

[6] The second condition is negligible in KDD, as it follows directly from minsupp > 0 %.

[7] In general, there may be edges which do not represent any rule in the Luxenburger basis. Consider for instance *minconf* = 7/10. In this case, the two lowest edges would not stand for a valid approximate rule, and would hence not be labelled.

$J := \{A\}$. The smallest frequent closed itemset containing B is $I' := \{B, E\}$ and the smallest one containing A and B is $J' := \{A, B, C, E\}$. In the diagram, I' and J' are always represented by the largest concepts which are below all attributes in I and $I \cup J$, resp. Between the two concepts we find the path $I_1 := I'$, $I_2 := \{B, C, E\}$, and $I_3 := J'$. Hence $supp(B \to A) = supp(J') = 0.4 \geq minsupp$ and $conf(B \to A) = conf(I_1 \to I_2) \cdot conf(I_2 \to I_3) = 3/4 \cdot 2/3 = 2/4 \geq minconf$, which means that the rule holds.

4 Algorithms for Computing the Bases

The algorithms presented in this paper assume that the iceberg concept lattice is already computed. There are several algorithms for computing iceberg concept lattices: the algorithm Close for strongly correlated data [31], the algorithm A-Close for weakly correlated data [30], the algorithms CLOSET [32], ChARM [43], and TITANIC [38,39]. The algorithm PASCAL [6] computes all (closed and non-closed) frequent itemsets, but can be upgraded to determine also their closures with almost no additional computation time by using the fact that, for $I \subseteq M$,

$$h(I) = I \cup \{m \in M \setminus I \mid supp(I) = supp(I \cup \{m\})\} \ .$$

When the iceberg concept lattice is computed, then the Duquenne–Guigues basis and finally the Luxenburger basis are computed.

4.1 Generating the Duquenne-Guigues Basis for Exact Association Rules with Gen-FDG

In this section, we present an algorithm that determines the Duquenne–Guigues basis using the iceberg concept lattice. This algorithm (which has not been presented before) implements Definition 4. As it needs to know the closure of frequent itemsets, it is best applied after an algorithm like PASCAL with the modification mentioned above, ChARM, or CLOSET.

The pseudo-code is given in Algorithm 1. The algorithm takes as input the sets FI_i, $1 \leq i \leq k$, containing the frequent itemsets and their support, and the sets FC_i, $0 \leq i \leq k$, containing the frequent closed itemsets and their support. It first computes the frequent pseudo-closed itemsets iteratively (steps 2 to 17). In steps 2 and 3, the empty set is examined. (It must be either a closed or a pseudo-closed itemset by definition.) The loop from step 4 to 17 is a direct implementation of Definition 4 for the frequent pseudo-closed itemsets. The frequent pseudo-closed i-itemsets, their closure and their support are stored in FP_i. They are used to generate the set $\mathcal{L}$ of implications of the Duquenne-Guigues basis for exact association rules DG (step 18).

The set of infrequent pseudo-closed itemsets is determined in steps 19 to 21 using the function $\mathcal{L}^*$-closure (Algorithm 2). This function uses the fact that, for a given closure system, the set of all closed or pseudo-closed sets forms again a closure system [14]. Hence one can generate all closed sets and pseudo-closed sets

Algorithm 1 Generating the Duquenne-Guigues basis with Gen-FDG.

1) $\mathcal{L} \leftarrow \{\}$;
2) **if** $(FC_0 = \{\})$ **then** $FP_0 \leftarrow \emptyset$;
3) **else** $FP_0 \leftarrow \{\}$;
4) **for** $(i \leftarrow 1; i \leq k; i{+}{+})$ **do begin**
5) $\quad FP_i \leftarrow FI_i \setminus FC_i$;
6) $\quad$ **forall** $L \in FP_i$ **do begin**
7) $\quad\quad pseudo \leftarrow true$;
8) $\quad\quad$ **forall** $P \in FP_j$ with $j < i$ **do begin**
9) $\quad\quad\quad$ **if** $(P \subset L)$ **and** $(P.\text{closure} \not\subseteq L)$
10) $\quad\quad\quad$ **then do begin**
11) $\quad\quad\quad\quad pseudo \leftarrow false$;
12) $\quad\quad\quad\quad FP_i \leftarrow FP_i \setminus \{L\}$;
13) $\quad\quad\quad$ **endif**
14) $\quad\quad$ **end**
15) $\quad\quad$ **if** $(pseudo = true)$ **then** $L.\text{closure} \leftarrow \min_{\subseteq}(\{C \in FC_{j>i} \mid L \subseteq C\})$;
16) $\quad$ **end**
17) **end**
18) **forall** $P \in \bigcup_{i=1}^{n} FP_i$ **do** $\mathcal{L} \leftarrow \mathcal{L} \cup \{P \rightarrow (P.\text{closure} \backslash P)\}$;
19) $IP \leftarrow \emptyset$;
20) **forall** $L \in MI$ **do** $IP \leftarrow IP \cup \{\mathcal{L}^*\text{-closure}(I)\}$;
21) $IP \leftarrow \min_{\subseteq} IP$;

iteratively by using the corresponding closure operator $\mathcal{L}^*$-closure$(Z) := \bigcup_{i=0}^{\infty} Z_i$ with $Z_0 := Z$ and $Z_{i+1} := Z_i \cup \bigcup\{Y | X \rightarrow Y \in \mathcal{L}, X \subset Z_i\}$ [14]. The set $\mathcal{L}$ of implications has the form $\mathcal{L} = \{X_1 \rightarrow Y_1, \ldots, X_n \rightarrow Y_n\}$.

4.2 Generating the Luxenburger Basis for Approximate Association Rules with Gen-LB

The pseudo-code generating the Luxenburger basis for approximate association rules is presented in Algorithm 3. The algorithm takes as input the sets FC_i, $0 \leq i \leq k$, containing the frequent closed itemsets and their support. The output of the algorithm is the Luxenburger basis for approximate association rules LB. The algorithm iteratively considers all frequent closed itemsets $L \in FC_i$ for $2 \leq i \leq k$. It determines which frequent closed itemsets $L' \in \bigcup_{j<i} FC_j$ are covered by L and generates association rules of the form $L' \rightarrow L \setminus L'$ that have sufficient confidence. During the i^{th} iteration, each itemset L in FC_i is considered (steps 3 to 13). For each set FC_j, $1 \leq j < i$, a set S_j containing all frequent closed j-itemsets in FC_j that are subsets of L is created (step 4). Then, all these subsets of L are considered in decreasing order of their sizes (steps 5 to 12). For each of these subsets $L' \in S_j$, the confidence of the approximate association rule $r := L' \rightarrow L \setminus L'$ is computed (step 7). If the confidence of r is sufficient, r is inserted into LB (step 9) and all subsets L'' of L' are removed from S_l, for $l < j$ (step 10). At the end of the algorithm, the set LB contains

Algorithm 2 Function $\mathcal{L}^*$-closure reads X and returns its $\mathcal{L}^*$-closure $\mathcal{L}^*(X)$.

```
1)  Y ← X;
2)  for (i ← 1; i = n; i++) do i.used← false;
3)  repeat
4)      changed ← false;
5)      If Subsets(IP, Y) /=∅ then begin Y ← M; changed← true end
6)      else for (i ← 1; i ≤ n; i++) do
7)                          if X_i ⊂ Y then begin Y ← Y ∪ Y_i; changed← true end
8)  until not changed;
9)  return Y
```

Algorithm 3 Generating the Luxenburger basis with Gen-LB.

```
1)  LB ← {};
2)  for (i ← 2; i ≤ k; i++) do begin
3)     forall L ∈ FC_i do begin
4)        for (j ← 0; J < i; j++) do S_j ← Subsets(FC_j, L);
5)        for (j ← i − 1; J ≥ 1; j−−) do begin
6)           forall L' ∈ S_j do begin
7)              conf ← L.support / L'.support;
8)              if (conf ≥ minconf)
9)                 then LB ← LB ∪ {(L' → (L \ L'), L.support, conf)};
10)             for (l ← j; l ≥ 1; l−−) do S_l ← S_l \ Subsets(S_l, L');
11)          end
12)       end
13)    end
14) end
```

all rules of the Luxenburger basis for approximate association rules. The proof of the correctness of the algorithm is given in [27].

5 Experimental Results

We have preformed several experiments on synthetic and real data. The characteristics of the datasets used in the experiments are given in Table 1. These datasets are the T10I4D100K synthetic dataset that mimics market basket data,[8] the C20D10K and the C73D10K census datasets from the PUMS sample file,[9] and the MUSHROOMS dataset describing mushroom characteristics.[10] In all experiments, we attempted to choose significant minimum support and confidence threshold values. We varied these thresholds and, for each couple of values, we analyzed rules extracted in the bases.

Number of Rules. Table 2 compares the size of the Duquenne-Guigues basis for exact rules with the number of all exact association rules, and the size of the

[8] http://www.almaden.ibm.com/cs/quest/syndata.html
[9] ftp://ftp2.cc.ukans.edu/pub/ippr/census/pums/pums90ks.zip
[10] ftp://ftp.ics.uci.edu/~cmerz/mldb.tar.Z

Table 1. Datasets.

Name	Number of objects	Average size of objects	Number of items
T10I4D100K	100,000	10	1,000
MUSHROOMS	8,416	23	127
C20D10K	10,000	20	386
C73D10K	10,000	73	2,177

Table 2. Number of exact and approximate association rules compared with the number of rules in the Duquenne-Guigues and Luxenburger bases.

Dataset (Minsupp)	Exact rules	D.-G. basis	Minconf	Approximate rules	Luxenburger basis
T10I4D100K (0.5%)	0	0	90%	16,269	3,511
			70%	20,419	4,004
			50%	21,686	4,191
			30%	22,952	4,519
MUSHROOMS (30%)	7,476	69	90%	12,911	563
			70%	37,671	968
			50%	56,703	1,169
			30%	71,412	1,260
C20D10K (50%)	2,277	11	90%	36,012	1,379
			70%	89,601	1,948
			50%	116,791	1,948
			30%	116,791	1,948
C73D10K (90%)	52,035	15	95%	1,606,726	4,052
			90%	2,053,896	4,089
			85%	2,053,936	4,089
			80%	2,053,936	4,089

Luxenburger basis for approximate rules with the number of all approximate rules. In the case of weakly correlated data (T10I4D100K), no exact rule is generated. The reason is that in such data all frequent itemsets are frequent closed itemsets. However, the Luxenburger basis is relatively small compared to the number of all rules, since only immediate neighbors with respect to the subset order (and not arbitrary pairs of sets) are considered. In the case of strongly correlated data (MUSHROOMS, C20D10K and C73D10K), the ratio between the size of the bases to the number of all rules which hold is much smaller than in the weekly correlated case, because here only few of the frequent itemsets are closed and have to be considered.

Relative Performance. Our experiments also show that in all cases the execution time of Gen-FDG and Gen-LB are insignificantly small compared to those of the computation of the iceberg concept lattice, since both algorithms need not access the database. We can conclude that without additional computation time (com-

pared to other approaches, like e. g. Apriori) our approach not only computes all frequent closed itemsets but also the two bases described in Section 2.

6 Outlook

In this paper, we introduced bases which significantly reduce the number of association rules presented to the user without losing any information; and provided algorithms for computing them. This work is currently extended in different directions: *Integrating reduction methods.* Templates, as defined in [4,21], can directly be used for extracting all association rules matching some user specified patterns from the bases. Information in taxonomies and ontologies associated with the dataset can also be integrated in the process as proposed in [17,34] for extracting bases for generalized (multi-level) association rules. Integrating item constraints [8,26,35] and statistical measures [10] in the generation of bases requires further work.

Integration of association rule visualization in Conceptual Information Systems. Using the technique of conceptual scaling, Conceptual Information Systems present the information contained in large databases to the user in conceptual hierarchies of a manageable size [40,36,18]. We work on exploiting this visualization techniques for presenting also association rules to the user.

Supporting the creation of new concepts in Description Logics. In Description Logics, currently approaches are discussed to support the domain expert in creating new concepts which regroup more specific similar concepts [3]. Those approaches extend the partial order of the concepts in the terminology to a lattice and suggest new concepts to the user. Since the more specific concepts are often defined incoherently, the user is often interested in only approximate relationships between those concepts, and on a general level only. It is planned to adapt the bases and the algorithms presented in this paper to that task.

References

1. R. Agrawal, T. Imielinski, A. Swami. Mining association rules between sets of items in large databases. *Proc. SIGMOD Conf.*, 1993, 207–216
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proc. VLDB Conf.*, 1994, 478–499 (Expanded version in IBM Report RJ9839)
3. F. Baader, R. Molitor: Building and structuring Description Logic knowledge bases using least common subsumers and Concept Analysis. In: B. Ganter, G. W. Mineau (eds.): *Conceptual Structures: Logical, Linguistic, and Computational Issues.* Proc. ICCS 2000. LNAI **1867**, Springer, Heidelberg 2000, 292–305
4. E. Baralis and G. Psaila. Designing templates for mining association rules. *Journal of Intelligent Information Systems* 9(1), 1997, 7–32
5. Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, L. Lakhal: Mining minimal non-redundant association rules using frequent closed itemsets. In: J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.–K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, P. J. Stuckey (Eds.): *Computational Logic — CL 2000.* Proc. 1st Intl. Conf. on CL (6th Intl. Conf. on Database Systems). LNAI **1861**, Springer, Heidelberg 2000, 972–986

6. Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, L. Lakhal: Mining Frequent Patterns with Counting Inference. *SIGKDD Explorations* **2**(2), Special Issue on Scalable Algorithms, 2000, 66–75
7. R. J. Bayardo. Efficiently mining long patterns from databases. *Proc. SIGMOD Conf.*, 1998, 85–93
8. R. J. Bayardo, R. Agrawal, D. Gunopulos. Constraint-based rule mining in large, dense databases. *Proc. ICDE Conf.*, 1999, 188–197
9. K. Becker, G. Stumme, R. Wille, U. Wille, M. Zickwolff: Conceptual Information Systems discussed through an IT-security tool. In: R. Dieng, O. Corby (Eds.): *Knowledge Engineering and Knowledge Management. Methods, Models, and Tools.* Proc. EKAW '00. LNAI **1937**, Springer, Heidelberg 2000, 352–365
10. S. Brin, R. Motwani, C. Silverstein: Beyond market baskets: Generalizing association rules to correlation. *Proc. SIGMOD Conf.*, 1997, 265–276
11. S. Brin, R. Motwani, J. D. Ullman, S. Tsur: Dynamic itemset counting and implication rules for market basket data. *Proc. SIGMOD Conf.*, 1997, 255–264
12. V. Duquenne, J.-L. Guigues: Famille minimale d'implication informatives résultant d'un tableau de données binaires. *Mathématiques et Sciences Humaines* 24(95), 1986, 5–18
13. U. M. Fayyad, G. Piatetsky–Shapiro, P. Smyth, R. Uthurusamy (eds.): Advances in knowledge discovery and data mining. AAAI Press, Cambridge 1996
14. B. Ganter, K. Reuter: Finding all closed sets: A general approach. *Order*. Kluwer Academic Publishers, 1991, 283–290
15. B. Ganter, R. Wille: *Formal Concept Analysis: Mathematical Foundations*. Springer, Heidelberg 1999
16. A. Grosskopf and G. Harras: Eine TOSCANA-Anwendung für Sprechaktverben des Deutschen. In: G. Stumme and R. Wille (eds.), Begriffliche Wissensverarbeitung: Methoden und Anwendungen. Springer, Berlin-Heidelberg-New York 2000.
17. J. Han, Y. Fu: Discovery of multiple-level association rules from large databases. *Proc. VLDB Conf.*, 1995, 420–431 1995.
18. J. Hereth, G. Stumme, U. Wille, R. Wille: Conceptual Knowledge Discovery and Data Analysis. In: B. Ganter, G. W. Mineau (eds.): *Conceptual Structures: Logical, Linguistic, and Computational Issues*. Proc. ICCS 2000. LNAI **1867**, Springer, Heidelberg 2000, 421–437
19. J. Hipp, A. Myka, R. Wirth, U. Güntzer: A new algorithm for faster mining of generalized association rules. LNAI **1510**, Springer, Heidelberg 1998
20. U. Kaufmann: Begriffliche Analyse über Flugereignisse – Implementierung eines Erkundungs- und Analysesystems mit TOSCANA. Diplomarbeit, FB4, TU Darmstadt 1996.
21. M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, A. I. Verkamo: Finding interesting rules from large sets of discovered association rules. *Proc. CIKM Conf.*, 1994, 401–407
22. M. Luxenburger: Implications partielles dans un contexte. *Mathématiques, Informatique et Sciences Humaines*, 29(113), 1991, 35–55
23. M. Luxenburger: Partial implications. Part I of *Implikationen, Abhängigkeiten und Galois Abbildungen*. PhD thesis, TU Darmstadt. Shaker, Aachen 1993
24. H. Mannila, H. Toivonen: Multiple uses of frequent sets and condensed representations (Extended abstract). *Proc. KDD 1996*, 189–194
25. R. Meo, G. Psaila, S. Ceri: A new SQL-like operator for mining association rules. *Proc. VLDB Conf.*, 1996, 122–133
26. R. T. Ng, V. S. Lakshmanan, J. Han, A. Pang: Exploratory mining and pruning optimizations of constrained association rules. *Proc. SIGMOD Conf.*, 1998, 13–24

27. N. Pasquier: *Data Mining : algorithmes d'extraction et de réduction des règles d'association dans les bases de données.* PhD thesis. Université Blaise Pascal, Clermont-Ferrand II, 2000
28. N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal: Pruning closed itemset lattices for association rules. *Proc. 14ièmes Journées Bases de Données Avancées (BDA '98)*, Hammamet, Tunisie, 177–196
29. N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal: Closed set based discovery of small covers for association rules. *Proc. 15èmes Journées Bases de Données Avancées*, Bordeaux, France, 25–27 October 1999, 361–381
30. N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal: Discovering frequent closed itemsets for association rules. *Proc. ICDT Conf.*, 1999, 398–416
31. N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal: Efficient mining of association rules using closed itemset lattices. *Journal of Information Systems*, 24(1), 1999, 25–46
32. J. Pei, J. Han, R. Mao: CLOSET: An efficient algorithm for mining frequent closed itemsets. *Proc. ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery 2000*, 21–30
33. M. Roth-Hintz, M. Mieth, T. Wetter, S. Strahringer, B. Groh, R. Wille: Investgating SNOMED by Formal Concept Analysis. Preprint, FB4, TU Darmstadt 1997.
34. R. Srikant, R. Agrawal: Mining generalized association rules. *Proc. VLDB Conf.*, 1995, 407–419
35. R. Srikant, Q. Vu, R. Agrawal: Mining association rules with item constraints. *Proc. KDD Conf.*, 1997, 67–73
36. G. Stumme, R. Wille, U. Wille: Conceptual Knowledge Discovery in Databases Using Formal Concept Analysis Methods. In: J. M. Żytkow, M. Quafofou (eds.): *Principles of Data Mining and Knowledge Discovery.* Proc. 2nd European Symposium on PKDD '98, LNAI **1510**, Springer, Heidelberg 1998, 450–458
37. G. Stumme: *Conceptual Knowledge Discovery with Frequent Concept Lattices.* FB4-Preprint **2043**, TU Darmstadt 1999
38. G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, L. Lakhal: Fast Computation of Concept Lattices Using Data Mining Techniques. *Proc. 7th Intl. Workshop on Knowledge Representation Meets Databases*, Berlin, 21–22. August 2000. CEUR-Workshop Proceeding. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/
39. G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, L. Lakhal: Computing Iceberg Concept Lattices with TITANIC. *J. on Knowledge and Data Engineering.* (submitted)
40. F. Vogt, R. Wille: TOSCANA – A graphical tool for analyzing and exploring data. LNCS **894**, Springer, Heidelberg 1995, 226–233
41. R. Wille: Restructuring lattice theory: an approach based on hierarchies of concepts. In: I. Rival (ed.). *Ordered sets.* Reidel, Dordrecht–Boston 1982, 445–470
42. M. J. Zaki, M. Ogihara: Theoretical Foundations of Association Rules, *3rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*, Seattle, WA, June 1998, 7:1–7:8
43. M. J. Zaki, C.–J. Hsiao: ChARM: An efficient algorithm for closed association rule mining. Technical Report 99–10, Computer Science Dept., Rensselaer Polytechnic Institute, October 1999
44. M. J. Zaki: Generating non-redundant association rules. *Proc. KDD 2000.* 34–43

Comparing Two Models for Software Debugging*

Markus Stumptner[1], Dominik Wieland[2], and Franz Wotawa[2]**

[1] University of South Australia, School of Computer and Information Science, Mawson Lakes Boulevard 1, 5092 Mawson Lakes SA, Australia, mst@cs.unisa.edu.au
[2] Technische Universität Wien, Institut für Informationssysteme, Database and Artificial Intelligence Group, Favoritenstraße 9–11, A-1040 Wien, Austria, {wieland,wotawa}@dbai.tuwien.ac.at

Abstract. This paper extends previous work on the representation and analysis of Java programs for diagnosis in a new direction by providing a description and analysis of the issues arising from handling object references in dependency-based models of Java programs. We empirically compare dependency-based models with a value-based model using a set of example programs in terms of required user interaction (questions put to the user) and examine and incorporate specific interesting error categories. Apart from being based on experience with an actual implementation of the various models, the model extensions and analysis deal with aliasing, an issue that the programming language community has been examining for a long time, and that is also crucial to object-orientedness.

1 Introduction

Detecting, locating, and repairing faults in software is a difficult and time consuming task. Detecting an incorrect behavior of a given program is done by using testing techniques (e.g., [1]) or formal verification methods, e.g., model checking [3]. Whereas much effort has been made on test theory, test methodology, and algorithms for automatic test-case generation, somewhat less work has been published on locating and repairing software faults. Because debugging is not only performed in the implementation and test phases of a project, but also in maintenance, saving debugging time naturally results in saving time and money over the whole product life cycle. Especially in maintenance, where the original developers may no longer be involved, debugging is very costly. An automated debugger for locating and fixing faults can help in such a situation.

Automatic debugging approaches introduced in the past include program slicing [22, 23], algorithmic debugging [19], dependency-based techniques [13,12,11], probability-based methods [2], and others. An overview of automatic debugging techniques can be found in [6]. These traditional approaches are either specific to a programming language, use specialized algorithms, or require explicit user-interaction to locate a bug. In order to overcome these drawbacks and to improve the results of the abovementioned approaches, the use of model-based diagnosis (MBD) for debugging was suggested [4]. Model-based diagnosis [18,5] provides a general theory for diagnosis that has sucessfully been applied

* This work was partially supported by Austrian Science Fund projects P12344-INF and N Z29-INF.

** Authors are listed in alphabetical order

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 351–365, 2001.

to various engineering areas. Apart from the diagnosis of technical systems, e.g., [14, 24], MBD has been used for knowledge bases [20,7] and other less technical domains, e.g., in ecology [10].

In this paper we build on the results obtained by the MBD community, especially as its application to the localisation of software bugs is concerned ([4] and more relevantly [9, 21]). Section 2 gives a brief description of the basics of MBD and shows how it can be used for fault localisation in programs. Section 3 presents some general thoughts about the modeling of Java programs and lists a sample program, which will be used throughout this paper to describe various model properties. Section 4 describes how a functional dependency model (FDM) of Java programs can automatically be derived from the program's source code. Furthermore, the transformation into a simplified functional-dependency model and the handling of object references (and therefore aliasing) are discussed. Section 5 shows how semantic information can be incorporated into the model by introducing a value-based model (VBM). Such a model allows for better results at the cost of an increased debugging time. Section 6 describes how the two models can be used to actually locate faults in programs. Furthermore, it presents empirical results produced by the `jade` prototype debugger. The `jade` debugging tool is a model-based debugger for Java programs featuring both, the use of functional dependency and value-based models. The diagnosis engine used by `jade` is based on Reiter's algorithm [18]. The `jade` user interface is designed to guide the user through the code in an optimal fashion, with optimality defined as minimizing the user interaction. An analysis of the debugging potentials of our approach and some future extentions conclude this work.

2 Model-Based Diagnosis and Debugging

The model-based approach is based on the notion of providing a representation of the correct behavior of a technical system. By describing the structure of a system and the function of its components, it is possible to ask for the reasons why the desired behavior was not achieved. In the diagnosis community, the model-based approach has achieved wide recognition due to the following advantages:

- once an adequate model has been developed for a particular domain, it can be used to diagnose different actual systems of that domain
- the model can be used to search for single or multiple faults in the system without alteration
- different diagnosis algorithms can be used for a given model
- the existence of a clear formal basis for judging and computing diagnoses

Using the standard consistency-based view as defined by Reiter [18], a diagnosis system can be formally seen as a tuple $(SD, COMP)$ where SD is a logical theory sentence modeling the behavior of the given system (in our case the program to be debugged), and $COMP$ a set of components, i.e., statements or expressions. A diagnosis system together with a set of observations OBS, i.e., a test-case, forms a diagnosis problem. A diagnosis Δ, i.e., a bug candidate, is a subset of $COMP$, with the property that the assumption that all statements (expressions) in Δ are incorrect, and the rest of the statements (expressions) is correct, should be consistent with SD and OBS. Formally,

Δ is a diagnosis iff $SD \cup OBS \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \cup \{AB(C) | C \in \Delta\}$ is consistent. A component not working as expected, i.e., a statement (expression) containing a bug, is represented by the predicate $AB(C)$.

The basis for this is that an incorrect output value (where the incorrectness can be observed directly or derived from observations of other signals) cannot be produced by a correctly functioning component with correct inputs. Therefore, to make a system with observed incorrect behavior consistent with the description and avoid a contradiction, some subset of its components must be assumed to work incorrectly. In practical terms, one is interested in finding minimal diagnoses, i.e., a minimal set of components whose malfunction explains the misbehavior of the system (otherwise, one could explain every error by simply assuming every component to be malfunctioning). Basic properties of the approach as well as algorithms for efficient computation of diagnoses are described in [18].

The principles of model-based debugging are depicted in Figure 1. The program, in our case written in Java, is compiled into an internal representation. From this representation (together with a set of *model fragments*) a converter computes logical models for diagnosis. Model fragments represent a logical description of parts of a model, e.g., the behavior description of functions. Such knowledge has to be derived from the programming language semantics. For Java for example, a value-based model requires the model fragments of all basic functions and types of statements, e.g., for arithmetic functions, behaviors (e.g., $Nab(C) \rightarrow out(C) = in_1(C) + in_2(C)$ for the **+** operator) must be defined. The predicate Nab stands for *not abnormal*, saying that a function or statement is not responsible for an incorrect behavior. After building the model, which is done automatically, the model together with the specified behavior of the program, e.g., test-cases, is used by the diagnosis engine for finding bug candidates. The candidates can be further discriminated by adding additional knowledge, i.e., values of variables at specific locations within the program. The selection of the variable and location is done by a measurement selection algorithm. The information about the value must be delivered by the user (or another oracle). The remaining candidates provide a link back to the original source code.

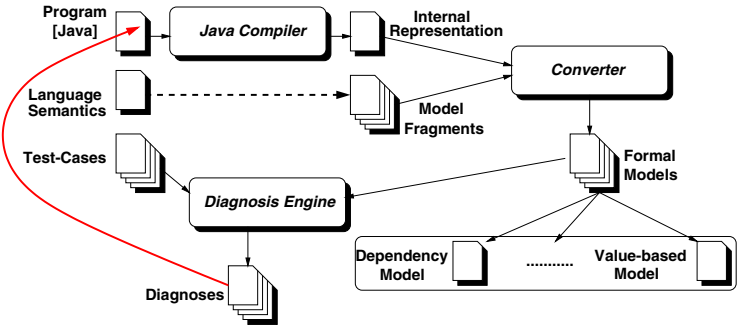


Fig. 1. MBD in software debugging

```
class Point {
      int x;
      int y;
      Point(int x, int y) {
1.       this.x = x;
2.       this.y = y; }
      Point plus(Point p) {
1.       return new Point(x+p.x, y+p.y); }

      public static void test() {
         Point p1, p2;
1.       p1 = new Point(0,0);
2.       p2 = new Point(2,3);
3.       p1.x = 1;
4.       p1.y = 2;
5.       p2 = p1.plus(p2); } }
```

Fig. 2. Example program *Point.java*

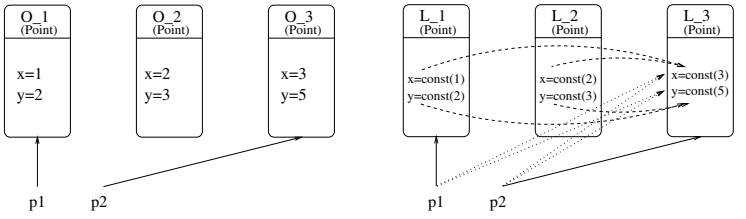
3 Modeling for Debugging

To demonstrate the modeling process and the properties of the resulting models we implement a short example program which provides the basic data structures and functionality of a 2-dimensional point (see Figure 2).

Note that variable declarations are not counted as statements, because no diagnosis components can arise from them. We classify the entities in the program in terms of

- The set $\mathcal{C}$ of *constants* (e.g., 0, 1, 2). Although constants in Java are not objects, they can logically be seen as objects with a fixed content of primitive type. Each occurrence of such a constant has to be considered separately, but we omit indexes for simplicity in the paper.
- The set $\mathcal{L}$ of memory *locations* which represent the stored state of Java objects. A location is a placeholder for objects that would be produced dynamically during runtime by constructor calls [11]. Their internal state is user defined and alterable (in our example 3 locations of type *Point* are created), but it is important to note that locations, which are introduced during static analysis of the functional dependencies, do not represent individual objects, but instead are generic representations covering all objects that might be assigned to a particular variable occurrence. In the value-based model explained later, individual objects are used instead.
- The set $\mathcal{V}$ of *variables* (including class and instance fields) can be of primitive or of reference type. Primitive variables (e.g. x and y above) hold values, reference variables (e.g. *p1* and *p2*) reference objects, i.e. they point at a particular memory location.

The state of a Java system at any time during execution can be specified by defining the three sets of system components given in the list above. Figure 3 shows the state of our example program after the execution of all five statements of method *test()*. We can see the current run time state of three objects (memory locations) given by their instance fields which are all assigned to constants. The two variables of reference type, *p1* and *p2* reference the first and third location, respectively. The object at location 2 is not referenced by any of the system's variables and will be eliminated by the garbage collector. In the dependency-based view, the values of the instance variables of *p2* are recorded as being derived from the values of those of location 2 (and location 1).



(a) Value-based (b) Dependency-based

Fig. 3. Representing objects

The following sections show how a Java system as defined above can be modeled for debugging using a static functional dependency model that for the first time incorporates object references (in Section 4) and a dynamic value-based model (Section 5).

4 The Dependency-Based Model

A dependency model for Java programs was presented in [15], defining a functional dependency (x_i, M_{x_i}), to express the fact that the value of the i-th occurrence of variable x in the current method depends on the values of the variables (actually, variable occurrences) in M_{x_i} if their values were used to compute the value stored by x_i (data dependency) or if they were used in selection or loop conditions that could have influenced the computation of that value (control dependency). In this paper, we describe two extensions of that model. The central issue in both is the explicit consideration of the semantics of object references, allowing a treatment of aliasing situations, which are generally excluded in classical dependency research (e.g, [17]), but are quite relevant in Java due to the language's general reference semantics for object-valued variables.

4.1 The Detailed Functional Dependency Model

When creating the detailed functional dependency model (DFDM) of a particular method we consecutively transform each statement of the method in question. Functional dependency components (hereinafter FDs) arise from assignment statements or program structures, that themselves include assignments, such as method calls (side-effect FDs), selection statements (selection FDs) or loops (loop FDs). Note that there might exist more than one FD for one statement, e.g., side-effect, selection, or loop FDs.

An FD in the detailed model is of the form $FD(var, DEP)$, where *var* is the variable occurrence whose value is determined in the particular statement and therefore depends on the entities in DEP. *var* can be of the following form: *x* if *x* denotes a class variable or a local variable of the currently modeled block; 0::x if *x* stands for an instance field of the receiver of the method; *n::x* where *n*> 0 is a location representing a different object than the receiver. *x* denotes an instance field of this particular object. Note that in all 3 cases *x* can be of primitive or of reference type.

DEP is a tuple $DEP =< C, V, M, L >$, where $C \subseteq \mathcal{C}$ is a set of constant values and $V \subseteq \mathcal{V}$ is the set of variables influencing *var*, M is a set of method declarations

(i.e., of the methods possibly called in the process of computing the new value for *var*), and $L \subseteq \mathcal{L}$ is a set of locations representing those objects that might be referenced by *var*.

Method *test()* from the example is modelled by collecting all FDs arising from its five statements. These FDs can be seen in the following report, which is automatically generated by the `jade` modeling component:

```
% Jade System Description Global Report
% Statement 1: p1=new Point(0,0)
      FD ( '1::y_1' <= { consts(0), vars(), methods(Point::Point(int,int)), locs() } )
      FD ( '1::x_1' <= { consts(0), vars(), methods(Point::Point(int,int)), locs() } )
      FD ( 'p1_1' <= { consts(), vars(), methods(Point::Point(int,int)), locs(1) } )
% Statement 2: p2=new Point(2,3)
      FD ( '2::y_1' <= { consts(3), vars(), methods(Point::Point(int,int)), locs() } )
      FD ( '2::x_1' <= { consts(2), vars(), methods(Point::Point(int,int)), locs() } )
      FD ( 'p2_1' <= { consts(), vars(), methods(Point::Point(int,int)), locs(2) } )
% Statement 3: p1.x=1
      FD ( '1::x_2' <= { consts(1), vars('p1_1'), methods(), locs() } )
% Statement 4: p1.y=2
      FD ( '1::y_2' <= { consts(2), vars('p1_1'), methods(), locs() } )
% Statement 5: p2=p1.plus(p2)
      FD ( '3::x_1' <= { consts(), vars('p1_1','1::x_2','p2_1','2::x_1'), methods(Point::Point(int,int),Point::plus(Point)), locs() } )
      FD ( '3::y_1' <= { consts(), vars('p1_1','1::y_2','p2_1','2::y_1'), methods(Point::Point(int,int),Point::plus(Point)), locs() } )
      FD ( 'p2_2' <= { consts(), vars('p1_1'), methods(Point::Point(int,int),Point::plus(Point)), locs(3) } )
```

Six of the above FDs are side-effect components and imported into the model of *test()* via the calls of *Point(int x, int y)* and *plus(Point p)*, the others are directly derived from the assignment statements in *test()*. Note that since the FD model is static, its creation does not involve or require any information about a particular evaluation trace.

4.2 A Simplified Functional Dependency Model

The advantage of the DFDM which records locations and references separately is that a broad range of program bugs can be located at statement level, due to the distinction being made between memory locations and variables. However, the DFDM is difficult to read and understand, and the large size of FDs slows down diagnosis performance. Also, the user needs detailed knowledge about the underlying object structure when specifying an incorrect variable observation.

This section therefore introduces a simplified functional dependency model (SFDM), which can automatically be derived from the DFDM. It is easier to understand, includes only variables on the FDs' right-hand sides, and makes it easier for the user to specify observations. FDs in SFDM again have the simpler structure of [15], but due to being derived from DFDM, still incorporate the effects of locations.

The conversion from DFDM to SFDM is obtained by successively simplifying each FD of the DFDM. Consider a FD $d = (vo, DEP)$, where $DEP =< C, V, M, L >$, to be converted to a SFD $d' = (vo', DEP')$, where DEP' is the subset of V that contains all local and class variables and only those instance variables, which are defined for the method's owner class, i.e., variable occurrences of the form *n::y* for $n > 0$ are deleted.

For the variable occurrence on the left hand side, assume first that vo is of the form x or *0::x* (i.e., it is a class or local variable occurrence or instance variable occurrence of the analyzed class. Then, $vo' = vo$. Now, assume that *vo* is of the form *n::x* referring to $n \in L$. Then all (local, class, or instance) variable occurrences y which at the given

point within the program possibly reference n are resolved by introducing a new FD of the form (y, DEP') (with DEP' as above).

As a result, FDs for variable occurrences of reference type now no longer simply denote that reference. Instead the dependency directly refers to the locations and references representing the local state of the referenced object (since these locations are no longer explicitly present in the model).

The overall set of FDs occurring at run-time can now be covered by $\mathcal{C}$ and $\mathcal{V}'$. The locations $\mathcal{L}$ are no longer explicitly part of the model, but implicitly covered through $\mathcal{V}'$.

The resulting SFDM for method *test()* reads as follows:

```
FD ( 'p1_1' ⇐ { vars()})                    % Statement 1: p1=new Point(0,0);
FD ( 'p2_1' ⇐ { vars()})                    % Statement 2: p2=new Point(2,3);
FD ( 'p1_2' ⇐ { vars('p1_1')})              % Statement 3: p1.x=1;
FD ( 'p1_3' ⇐ { vars('p1_2')})              % Statement 4: p1.y=2;
FD ( 'p2_2' ⇐ { vars('p1_3','p2_1')}) % Statement 5: p2=p1.plus(p2);
```

These two FD models introduce the notion of aliasing, which describes any situation where two variables of reference type refer to two different object structures, which in turn have references to the same object in common. Consider the following code fragment using two variables p1, p2 of class Point:

```
1.  p2 = p1;
2.  p1.doubleXValue();
3.  output(p2.x);
```

Assume the call to method doubleXValue() is erroneous, and an incorrect value is observed in p2.x (which is the same as p1.x since they refer to the same Point object). The model from [15] would suggest statement 1 as a potential source of the error, but would omit statement 2. Both, the DFDM and the SFDM introduce a location l on which both p1 and p2 depend and express the change caused by the method call in line 2 in terms of a change to l.x, thus making the aliasing between the two variables explicit.

5 The Value-Based Model

In the value-based diagnosis model both expressions and statements are represented as diagnosis components, with the semantics of the expressions and statements described in terms of logical sentences. Components are connected if there is a flow of information between the corresponding expressions and statements, e.g., if a changes a variable v that is accessed in s and there is no assignment to v in between.

5.1 Description

Figure 4 shows the graphical representation of the model of example program *test*. The source code of line 1 is mapped to four diagnosis components: two components for the constants ($C1.2, C1.3$), one for the *new Point* method call ($C1.1$), and one ($C1$) that corresponds to the assignment statement. Instance variable accesses are also represented as components ($C3$ or $C4$) with two parameters: the variable storing the object, and the

name of the instance variable (see line 3 or 4). If variables are used in an expression, then they are mapped to a variable access component (see components corresponding to line 5). We refer to a component C as "correct" if it is assumed to be correct, i.e., $\neg AB(C)$ holds.

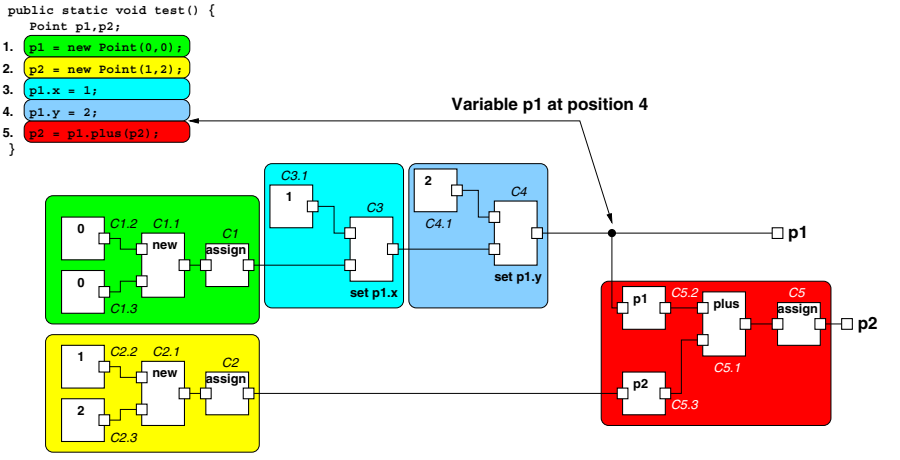


Fig. 4. Graphical representation of the value-based model

In the following behavior description, we use tuples of the form $[Id[Var_1 Val_1] \ldots [Var_n Val_n]]$ to represent objects with the unique object identifier Id, and the pairs $[Var_i\ Val_i]$ that map each instance variable Var_i to its value Val_i. Note that Val_i itself can be an object identifier.

Constants: Correct constants propagate their constant value, e.g., an integer or even an object, to the output port.

Variable access: This has one input, connected to the output of the last assignment having the same variable as target, and one output. A correct access propagates the input value to the output and known output values to the input.

Operators: An operator has an input port for each argument and one output. The inputs are connected to the components that correspond to subexpressions. The output is connected to a statement port, e.g., the expression port of an assignment, or to one input of the superexpression. The behavior of an operator is specified by the semantics of the corresponding statement. For example the Java *and* (`&&`) operator component has the following behavior: $\neg AB(C) \Rightarrow (in_1(C) \wedge in_2(C) = out(C))$

Method calls: The behavior of a correct method call is determined by the diagnosis model constructed from the method's source code to propagate the input parameter values to output parameters or return value.

Assignments: If assumed to be correct, assignments $v = e$ to locally declared variables propagate the single input value produced by expression e to the single output and vice versa. Assignments to instance variables (e.g., `p1.x = 3;`) have a second

input carrying the object. The output port is connected to components which access the variable. The correct behavior is to change only the instance variable V_i of the object $[Id\ [Var_1\ Val_1]\ \ldots [Var_n\ Val_n]]$ occurring on the input port of the variable. In the back propagation step the expected object is propagated from the output to the input port of the variable, and the value of the used instance variable is propagated to the output port of e.

Control statements: Loop components have as input connections the variables used inside the loop, and as output connections the variables altered by the loop, propagating values repeatedly (as needed) across the components of the body of the loop. Selection statements propagate values to either the **then** or **else** branch depending on condition evaluation.

Diagnoses are computed in the usual consistency-based manner [18]: A set $\Delta \subseteq COMP$ is a diagnosis iff $SD \cup OBS \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \cup \{AB(C) | C \in \Delta\}$ is consistent. $AB(C)$ indicates that a component C is behaving abnormally, and a correctly behaving component C is described by $\neg ab(C)$. In general we want to compute diagnoses which are subset-minimal.

To locate a bug in program *test*, assume that the call *test()* should compute the object $[?\ [x\ 1]\ [y\ 1]]$ to be stored in variable *p1*. Instead, *test* returns the object $[?\ [x\ 1]\ [y\ 2]]$ for variable *p1*. The test case produces four diagnoses: $\{AB(C1.1)\}, \{AB(C1)\}, \{AB(C4, 1)\}, \{AB(C4)\}$ which correspond to statements 1 and 4. If using a fault model for the *new Point* method and the assignment statement, we can rule out the diagnoses $\{AB(C1.1)\}$ and $\{AB)(C1)\}$ by propagating an empty object [? [x ?] [y ?]] to the output in case of an unknown fault AB. For more details of the value-based model see [16].

5.2 Refining Conditional Statements

The value-based model treats conditional statements loosely, i.e., in some cases too many diagnoses can be derived from the model. Consider the following program fragment with expected value for *result* being 0:

```
1.    x = 1;
2.    if (x < 1) {
3.       result = 1; }
4.    else { result = 1;} // 1 should be 0 !
```

With the original value-based model we derive diagnoses saying that either statement 1 or statement 2 (including the subblocks) causes the misbehavior. But it is obvious that the assignment in line 1 is not a single fault. This effect is due to the missing behavior in cases where the correct value of the condition expression of a conditional statement is known, which is the case when we assume that line 1 is faulty. A remedy is to extend the model of conditional statements:

1. If there is a variable x such that the **then** and **else** block compute different values for x and both values are different from the expected value, then raise a contradiction.

2. If there is a variable x such that **then** and **else** block compute the same value v for x, propagate v to the output.

Formally, the new model can be expressed as follows:

$\neg AB(C) \wedge cond(C) = true \Rightarrow (then_X(C) = out_X(C))$
$\neg AB(C) \wedge cond(C) = false \Rightarrow (else_X(C) = out_X(C))$
$\neg AB(C) \wedge cond(C) = unknown \wedge then_X(C) \not= out_X(C) \wedge else_X(C) \not= out_X(C) \Rightarrow \bot$
$\neg AB(C) \wedge cond(C) = unknown \wedge then_X(C) = else_X(C) \Rightarrow (out_X(C) = then_X(C))$

When using this model, line 1 is excluded as a diagnosis candidate.

6 Results

In this section we compare the dependency-based and value-based model, first in terms of the Java language constructs covered, then in terms of tests carried out with the `jade` debugger on both types of models using actual programs. Note that during all tests with FDMs a SFDM was used. Whereas a DFDM could produce more accurate results, its handling would be more difficult, especially as far as the exact specification of observations is concerned. We therefore use the DFDM as a generic model, which represents the basis for the SFDM and in future other possible dependency-based models. Finally we analyze some routes for future work.

Both models currently support classes and instances, static and instance methods and variables, method calls, polymorphism and aliasing. The value-based model currently does not allow for recursive functions. Both models support sequential code exclusively (no processes).

Since the value-based model expresses the full semantics of the covered language features, it can predict values by forward propagation (from inputs to outputs) or backward propagation (from outputs to inputs) whenever sufficient values are present on a component's connections. During debugging, it allows the user to specify intended actual values as observations instead of merely observing whether a value is correct or incorrect. Also, the value-based model's diagnosis granularity is lower and individual subexpressions can be identified as incorrect.

6.1 Using the Models for Debugging

The diagnosis models were implemented in the context of the `jade` debugger and tested on a Sun UltraSPARC II/360. Both models are automatically derived from the source code and compiled into a system description which can be represented as a set of logical sentences and thus be loaded into a standard theorem prover. The program is evaluated and the user is then asked to specify expected values (observations) for the connections of the system description. This means that when using the SFDM he has to state whether the value of a certain variable (directly derived from the observed in/out behaviour of the tested method) at a particular position within the source code has the correct value or not, and when using the VBM he can specify the intended values. The observations are converted into logical sentences and added to the debugger's system description. The user can continue specifying additional variable observations, in order to successively reduce the set of possible diagnoses and eventually find a single bug location. A measurement selection algorithm (the `jade` system uses a slightly modified version of [5]) automatically selects the measurement point, i.e. variable occurrence, whose evaluation

will reduce the number of diagnoses in an ideal way. After a certain number of such steps a single statement s containing the bug is found by the debugger. In this case the user has the following options:

1. If s contains subblocks (a selection or loop statement), the user can interactively guide the diagnosis process into the condition or one of the subblocks. If s is a loop, the user also has to select the first incorrect iteration from a displayed list. When stepping into a subblock, information from the enclosing statement s is propagated downwards. **2.** In all other cases the bug has been found at statement or expression level, and in the case of a method or constructor call, debugging can continue inside that call. **3.** Stop the debugging process, continue manually from the current diagnosis candidate list.

6.2 Empirical Results

The relative debugging potential of both models was tested on programs demonstrating simple variable dependencies (simulating a binary adder, numeric examples), making use of control structures (if and while statements), and finally multiple objects and instance fields together with linked lists and general processing (a small library application).

Table 5 shows the tested method (in which a single error has been installed), the total application code length (which determines the model size; esp. in the library example most of this code is also called by the buggy function), the number of statements in the tested method, the index of the buggy statement within the method (which can directly be used to compare the outcomes of the `jade` debugger tests with "manual" use of a debugger where the user steps through the code sequentially until the erroneous line is found), and finally the number of user interactions (variable setup, variable observation queries, program flow control queries) which are needed to exactly locate the bug[1]. The latter are defined as follows:

FD shows the total number of user interactions needed to locate a single error at statement level using the FD model.

V1 shows the performance of the VBM in statement level debugging. Selection statement queries are not needed with the VBM, because the system automatically guides the diagnosis process into the faulty subbranch. For each measurement selection query only one observation was specified.

V2 shows the number of interactions needed to locate the faulty *expression* using the VBM. In most cases it took one or two extra steps from statement level to expression level.

In general both debugging strategies perform better than a manual walkthrough. On average it takes 1.1 additional queries to find the solution at expression level. Therefore the VBM usually finds a solution at expression level quicker than the FDM at statement level.

In general the VBM achieves better results than the FDM, mainly due to the additional run-time information used by the VBM debugger, which allows for a more effective elimination of wrong diagnoses (see adder and numeric tests). With a growing number of variables and more complex dependency structures this effect is likely

[1] The average is included for overview purposes; it has no statistical significance since it depends on the specific selection of program and error examples.

Test	Method	LOC	Lines	Error	FD	V1	V2
1	*adder_f1*	71	17	4	3	3	4
2	*adder_f2*	71	17	4	3	3	4
3	*adder_f3*	71	17	7	3	2	3
4	*adder_f4*	71	17	7	3	2	3
5	*adder_f5*	71	17	7	3	3	4
6	*adder_f6*	71	17	12	5	4	5
7	*adder_f7*	71	17	12	5	4	5
8	*adder_f8*	71	17	11	5	3	4
9	*adder_f9*	71	17	11	7	4	5
10	*adder_f10*	71	17	14	5	2	3
11	*adder_f11*	71	17	14	5	3	4
12	*adder_f12*	71	17	16	5	5	6
13	*adder_f13*	71	17	12	5	4	5
14	*adder_f14*	71	17	9	5	4	5
15	*ifTest_f1*	109	5	4	4	3	4
16	*ifTest_f2*	109	5	4	3	2	3
17	*ifTest_f3*	109	9	8	4	4	4
18	*ifTest_f4*	109	6	5	5	2	3
19	*ifTest_f5*	109	8	8	5	2	3
20	*ifTest_f6*	109	12	10	5	3	4
21	*whileTest_f1*	119	12	6	3	3	3
22	*whileTest_f2*	119	12	10	6	6	7
23	*whileTest_f3*	119	13	4	2	3	3
24	*whileTest_f4*	119	13	6	3	3	5
25	*whileTest_f5*	119	13	10	10	8	9
26	*differntiat_f1*	98	4	3	4	4	5
27	*differntiat_f2*	98	4	3	4	4	5
28	*differntiat_f3*	98	4	4	4	2	4
29	*differntiat_f4*	98	4	4	4	2	4
30	*differntiat_f5*	98	4	4	4	2	4
31	*differntiat_f6*	98	4	4	4	2	4
32	*integrate_f1*	98	8	2	3	2	3
33	*integrate_f2*	98	8	2	2	2	2
34	*integrate_f3*	98	8	2	4	2	3
35	*integrate_f4*	98	8	6	7	7	10
36	*integrate_f5*	98	8	6	7	7	10
37	*trafficLight_f1*	106	15	12	9	7	8
38	*trafficLight_f2*	106	15	7	7	7	8
39	*library_f1*	215	31	24	8	8	8
Σ			575	342	237	143	186
Av.			11.98	7.12	4.94	3.67	4.77

Fig. 5. Debugging results from Java examples

to increase. If the error appears within an if statement, the VBM clearly outperforms the FDM (see if-tests), mainly due to the VBM debugger's ability to debug the body of a selection statement together with its enclosing block, whereas the FDM debugger performs a hierarchical diagnosis involving some overhead (control and setup GUIs). With loops, both models require extra queries on loop condition correctness (mainly to identify the first incorrect iteration) in combination with setup queries for the subblock's variables. This leads to more queries for both, reducing the advantages of the VBM. In the numeric examples the apparently high number of queries is somewhat misleading since the complex computations automatically produce good focusing and result in identifying detailed subexpressions quickly. With the library program both models seem to be equally good. Although the VBM uses more information than the SFDM, this does not pay off here because of the complex object structure of the application combined with the relatively simple operations possible in the example (mostly lookup operations concerning lists of books and customers which are highly dependent on the input values provided).

6.3 Discussion

The `jade` debugger utilizes a model-based diagnosis framework to incorporate multiple automatically derived models of programs into a standard debugger interface, allowing to switch between traditional step-based debugging and between models at the flick of a switch and providing iterative error location without the need for external specifications. The latter properties are the main distinction compared to formal verification methods

such as model checking, which requires a separate formal specification and provides counterexamples but no indication of the error locality.

Current implementation work includes better runtime for the VBM model, since loops with complex object structures such as large lists can result in diagnosis runtimes of several minutes. Howevere, most of the examples in the table were diagnosed in the 1-10 second range. Another goal is better visualization for indicating correctness of complex object structures.

In modeling, the identification and modeling of specific problem classes can be expected to lead to still more effective debugging. The dependency-based representation can be expected to scale up well to medium-sized programs (thousands of lines of code). (The somewhat simpler dependency-based representation described in [9] provided acceptable performance for programs with hundreds of thousands of lines of code.)

Faulty location structure. Either two variables point to different locations, i.e., objects, but should refer to the same object, or a variable refers to the same object as another variable but should point to a different object (possibly with the same content), as in this case:

```
1.    p1 = new Point(0,0);
2.    p2 = p1;              // Should be p1.copy()
3.    p1.x = 1;             // Expected results: p1=(1,0), p2=(0,0)
```

When using its simple functional dependency model

```
FD('p1_1' <- {vars()})                                          % Statement 1: p1 = new Point(0,0);
FD('p2_1' <- {vars('p1_1')})                                    % Statement 2: p2 = p1;
FD('p1_2' <- {vars('p1_1','p2_1')}, 'p2_2' <- {vars('p1_1')}) % Statement 3: p1.x = 1;
```

the debugger will list all statements as bug candidates. However, because statement 3 changes the value of *p1* and its value is correct at the end of the program, the statement can actually not be responsible for the faulty behavior. This additional candidate is caused by the dependency for 'p2_2'. If we eliminate this dependency from statement 3, the dependency-based model delivers the expected results, but can no longer deal with aliasing. One solution would be to use two different models, one with and one without aliasing. The best model to be used for a specific problem could then be chosen by the user (if a-priori knowledge about the problem exists) or by the use of given error class statistics. Similar observations hold for the DFDM and VBM.

Structural faults. The wrong variable in the program is accessed or changed, i.e., the dependency graph [8] of the program is not structurally equivalent to the dependency graph of the correct program. Such faults can be repaired by replacing the variable. The following fragment depicts an incorrect assignment target:

```
1.    int x=0, y=0 ;
2.    x = 2;
3.    x = 2; // Should be y=2. Expected results: x=2, y=2
```

For both models the debugger returns statement 1 as the only candidate but the expected bug cannot be located. In some cases, e.g., when line 3 reads `x=3` and we expect the outputs x=2 and y=3, the debugger is able to locate the bug, but in general

more powerful solutions need to be applied, such as the introduction of replacement fault modes for assignments [21].

The second subclass of structural faults is the wrong use of variables in expressions:

```
1.    x = 3;
2.    tmp = 2;
3.    y = 2 * tmp; // Should be y=2*x. Expected: x=3, y=6
```

Because no value for *tmp* is specified, the debugger would return statement 2 and 3 as diagnosis candidates regardless of model. With the VBM, we obtain the variable access *tmp* in line 3 as a single fault. Hence, this second class of structural faults can implicitly be handled by our models.

7 Conclusion

In this paper we have extended earlier work on dependency-based models of imperative programs by the description of dependencies for object references that allow the diagnosis of situations involving aliasing, and discuss the tradeoffs inherent in the models. We also present results gained from experimenting with the implementation that incorporates both, these models as well as, for the first time, the implementation of a value-based model of imperative programs. Both models represent an improvement over the earlier dependency-based model in terms of diagnosis discrimination. Runtime performance of the dependency-based models is satisfactory, while the value-based model still has to be improved in runtime performance. However the experiments have shown it to be superior to the dependency based models and stepwise manual debugging in terms of required user interaction.

References

1. Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.
2. Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, 38(3):31 – 41, 1995.
3. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
4. Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993.
5. Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
6. Mireille Ducassé. A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, Springer LNCS 749, pages 1–15, May 1993.
7. Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency based diagnosis of configuration knowledge-bases. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, June 1999.

8. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
9. Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.
10. Ulrich Heller and Peter Struss. Conceptual Modeling in the Environmental Domain. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 6, pages 147–152, Berlin, Germany, 1997.
11. Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.
12. Bogdan Korel. PELAS–Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, 1988.
13. Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, May 1989.
14. A. Malik, P. Struss, and M. Sachenbacher. Case studies in model-based diagnosis and fault analysis of car-subsystems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 1996.
15. Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java programs using a model-based approach. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, Scotland, 1999.
16. Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling Java Programs for Diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany, August 2000.
17. Mark Moriconi and Timothy Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Transactions on Software Engineering*, 16(9):980–992, September 1990.
18. Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
19. Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.
20. Markus Stumptner and Franz Wotawa. Model-based reconfiguration. In *Proceedings Artificial Intelligence in Design*, Lisbon, Portugal, 1998.
21. Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, pages 1074–1079, Stockholm, Sweden, August 1999.
22. Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.
23. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
24. Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, pages 1178–1185, 1997.

Inferring Implicit State Knowledge and Plans with Sensing Actions

Michael Thielscher

Dresden University of Technology

Abstract. An effective method is presented for deriving state knowledge in the presence of sensing actions. It is shown how conditional plans can be inferred with the help of a generalized concept of plan skeletons as search heuristics, which allow the planner to introduce conditional branching points by need.

1 Introduction

The problem of modeling sensing actions has gained much attention in the recent past as an important step for the development of extensive foundations for Cognitive Robotics. Several solutions to the technical Frame Problem have been generalized to reasoning about the knowledge of a robot and the effect of sensing, e.g., in the Situation Calculus [20] and the Fluent Calculus [23]. Based on general first-order logic, these approaches are sufficiently expressive to allow for modeling actions with knowledge preconditions, sensing of non-atomic properties, and deriving implicit knowledge. Moreover, to solve planning problems involving knowledge goals, the notion of conditional plans has been integrated [11,23] since it may be necessary to plan ahead different action sequences for different outcomes of sensing [14].

The expressiveness of general theories for conditional planning, on the other hand, raises the challenge to evolve inference algorithms that efficiently deal with the modality of knowledge. Most existing planning methods are tailored to restricted classes of planning problems, e.g., [9,7,2,16,11]. In particular, none of these systems can solve planning problems where knowledge follows *implicitly*: A well-known example is to determine acidity of a chemical solution by sensing the color of a Litmus strip [18]. The only existing system with a general solution to the Frame Problem for knowledge is [19] based on GOLOG [15]. However, this Prolog implementation is not meant for planning with sensing as it does not allow to *search* for suitable sensing actions. Rather, the user is supposed to provide GOLOG programs where all necessary sensing actions have been correctly planned. This restriction to plan verification applies to other existing approaches as well, such as [10].

In this paper, we present the foundations for an effective, fully automatic reasoning system capable of solving planning problems which require conditional plans and implicit knowledge and which may involve incomplete states, nondeterministic actions, and knowledge preconditions as well as knowledge goals.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 366–380, 2001.

Based on the recent solution to the Frame Problem for knowledge in the Fluent Calculus [23], our main technical result is a proof that, under reasonable assumptions, knowledge can be identified with incomplete state specifications. This theorem is applied to FLUX (the Fluent Calculus Executor)—a recent logic programming methodology for Cognitive Robotics [22] with similar motivations as GOLOG [15] but where state update axioms [21] are used to solve the inferential Frame Problem [3] and where constraints are used for encoding incomplete states.

Since conditional planning is a highly complex search problem, we also adapt the heuristics of nondeterministic robot programs of GOLOG [15] and develop a generalization which allows to search for plans in the presence of sensing. Conditionals occurring in the plan skeleton are evaluated at planning time only if the state knowledge suffices to do so; otherwise, a branching point is introduced, leading to a conditional plan by need. Prior to presenting the results, we give a brief introduction to the basic Fluent Calculus and FLUX.

2 The Fluent Calculus for Knowledge and Sensing

2.1 State Update Axioms

The basic Fluent Calculus combines, in pure classical logic, the Situation Calculus with a STRIPS-like solution to the representational and inferential Frame Problem [21]. The standard sorts ACTION and SIT (i.e., situations) are inherited from the Situation Calculus [13] along with the standard functions $S_0 : \text{SIT}$ and $Do : \text{ACTION} \times \text{SIT} \mapsto \text{SIT}$ denoting, resp., the initial situation and the successor situation after performing an action; furthermore, the standard predicate $Poss : \text{ACTION} \times \text{SIT}$ denotes whether an action is possible in a situation. To this the Fluent Calculus adds the sort STATE with sub-sort FLUENT along with the pre-defined functions $\emptyset : \text{STATE}$; $\circ : \text{STATE} \times \text{STATE} \mapsto \text{STATE}$; and $State : \text{SIT} \mapsto \text{STATE}$; denoting, resp., the empty state, the union of two states, and the state of the world in a situation. Based on this signature, the Fluent Calculus provides a rigorously logical account of the concept of a state being characterized by the set of fluents that are true in the state. The following foundational axioms serve this purpose. They are a suitable subset of the Zermelo-Fraenkel axioms, stipulating that function $\circ$ behaves like set union with $\emptyset$ as the empty set:[1]

$$
\begin{array}{ll}
z_1 \circ (z_2 \circ z_3) = (z_1 \circ z_2) \circ z_3 & \neg Holds(f, \emptyset) \\
z_1 \circ z_2 = z_2 \circ z_1 & Holds(f_1, f) \supset f = f_1 \\
z \circ z = z & Holds(f, z_1 \circ z_2) \supset Holds(f, z_1) \vee Holds(f, z_2) \\
z \circ \emptyset = z & (\forall f)\,(Holds(f, z_1) \equiv Holds(f, z_2)) \supset z_1 = z_2 \\
\multicolumn{2}{c}{(\forall \Phi)(\exists z)(\forall f)\,(Holds(f, z) \equiv \Phi(f))}
\end{array}
$$

[1] Free variables in formulas are assumed universally quantified. Variables of sorts ACTION, SIT, FLUENT, and STATE shall be denoted by the letters a, s, f, and z, resp. The function $\circ$ is written in infix notation.

where Φ is a second-order predicate variable of sort FLUENT and the macro *Holds* means that a fluent is contained in a state:

$$Holds(f, z) \stackrel{\text{def}}{=} (\exists z')\, z = f \circ z' \tag{1}$$

The very last one of the foundational axioms above stipulates the existence of a state for all possible combinations of fluents. A second macro, which reduces to (1), is used for fluents holding in situations:

$$Holds(f, s) \stackrel{\text{def}}{=} Holds(f, State(s))$$

Consider, e.g., the FLUENT terms $OnTable(x)$, $Acidic(x)$, $Carries(x)$, and $Red(y)$, denoting, resp., whether a chemical solution x is on the table, x is acidic, the robot carries x, and Litmus strip y is red.[2] The following incomplete state specification says that initially there are three chemical solutions A, B, and C on the table, litmus paper P is not red, the robot carries nothing, and either B or C is not acidic:

$$\begin{array}{l} Holds(OnTable(A), S_0) \wedge Holds(OnTable(B), S_0) \wedge Holds(OnTable(C), S_0) \\ \wedge \neg Holds(Red(P), S_0) \wedge (\forall x)\, \neg Holds(Carries(x), S_0) \\ \wedge [\neg Holds(Acidic(B), S_0) \vee \neg Holds(Acidic(C), S_0)] \end{array} \tag{2}$$

Assuming uniqueness of names for all fluents, the macro definitions and the foundational axioms imply that (2) is equivalent to

$$\begin{array}{l} (\exists z)\, (\, State(S_0) = OnTable(A) \circ OnTable(B) \circ OnTable(C) \circ z \\ \quad \wedge \neg Holds(Red(P), z) \wedge (\forall x)\, \neg Holds(Carries(x), z) \\ \quad \wedge [\neg Holds(Acidic(B), z) \vee \neg Holds(Acidic(C), z)] \\ \quad \wedge \neg Holds(OnTable(A), z) \wedge \neg Holds(OnTable(B), z) \\ \quad \wedge \neg Holds(OnTable(C), z)\,) \end{array} \tag{3}$$

The reader may notice that the constraints on sub-STATE z not only reflect the negated *Holds* statements of (2) but also the fact that neither of $OnTable(A)$, $OnTable(B)$, or $OnTable(C)$ re-occurs. This will allow to quickly infer the result of removing any of these fluents from $State(S_0)$ as a negative effect.

The Frame Problem is solved in the Fluent Calculus using so-called state update axioms, which specify the difference between the states before and after an action. The axiomatic characterization of negative effects, i.e., facts that become false, is given by this inductive abbreviation, which generalizes STRIPS-style update [4] to incomplete states:

$$z' = z - f \stackrel{\text{def}}{=} [z' \circ f = z \vee z' = z] \wedge \neg Holds(f, z')$$

$$\begin{array}{l} z' = z - (f_1 \circ \ldots \circ f_n \circ f_{n+1}) \stackrel{\text{def}}{=} \\ \quad (\exists z'')\, (z'' = z - (f_1 \circ \ldots \circ f_n) \wedge z' = z'' - f_{n+1}) \end{array}$$

[2] This scenario, which will be used throughout the paper, is a variation of an example first used in [18].

On this basis, the following is the general form of a state update axiom for a (possibly nondeterministic) action $A(\boldsymbol{x})$ with a bounded number of (possibly conditional) effects:

$$\begin{aligned} Poss(A(\boldsymbol{x}), s) \supset\ & (\exists \boldsymbol{y}_1)\,(\Delta_1 \wedge State(Do(A(\boldsymbol{x}), s)) = (State(s) \circ \vartheta_1^+) - \vartheta_1^-) \\ & \vee \ldots \vee \\ & (\exists \boldsymbol{y}_n)\,(\Delta_n \wedge State(Do(A(\boldsymbol{x}), s)) = (State(s) \circ \vartheta_n^+) - \vartheta_n^-) \end{aligned}$$

where the sub-formulas $\Delta_i(\boldsymbol{x}, \boldsymbol{y}_i, State(s))$ specify the conditions on $State(s)$ under which $A(\boldsymbol{x})$ has the positive and negative effects ϑ_i^+ and ϑ_i^-, resp. Both ϑ_i^+ and ϑ_i^- are STATE terms composed of fluents with variables among $\boldsymbol{x}, \boldsymbol{y}_i$. If $n = 1$ and $\Delta_1 \equiv True$, then action $A(\boldsymbol{x})$ does not have conditional effects. If $n > 1$ and the conditions Δ_i are not mutually exclusive, then the action is nondeterministic.

Consider, e.g., the ACTION terms $Take(x)$ and $Test(x, y)$ denoting, resp., the robot taking x off the table and testing x by inserting Litmus paper y. The effects of these two actions can be defined by these state update axioms:

$$\begin{aligned} & Poss(Take(x), s) \supset \\ & \quad State(Do(Take(x), s)) = (State(s) \circ Carries(x)) - OnTable(x) \\ & \\ & Poss(Test(x, y), s) \supset \\ & \quad [Holds(Acidic(x), s) \wedge State(Do(Test(x, y), s)) = State(s) \circ Red(y)] \vee \\ & \quad [\neg Holds(Acidic(x), s) \wedge State(Do(Test(x, y), s)) = State(s)] \end{aligned} \tag{4}$$

Put in words, taking x has the effect that the robot carries x and x is no longer on the table; and testing x with the help of Litmus paper y causes y to turn red if the solution is acidic, otherwise nothing changes. The action preconditions shall be defined by:

$$\begin{aligned} Poss(Take(x), s) &\equiv Holds(OnTable(x), s) \\ Poss(Test(x, y), s) &\equiv True \end{aligned}$$

Recall formula (3). The state update axiom for $Take(x)$ and the foundational axioms imply

$$\begin{aligned} (\exists z)\,(\, & State(Do(Take(A), S_0)) = OnTable(B) \circ OnTable(C) \circ z \circ Carries(A) \\ & \wedge \neg Holds(OnTable(A), z)\,) \end{aligned}$$

Besides the positive effect $Carries(A)$, the right hand side of the equation includes all fluents which are not affected by the action. Moreover, facts given in (3) as to which fluents do not hold in z apply to the new state just as well as it includes z. Thus all unchanged knowledge continues to hold without the need to apply extra inference steps.

2.2 FLUX

The programming language FLUX is a recent implementation of the Fluent Calculus based on Constraint Logic Programming [22]. Its distinguishing feature is to support incomplete states, which are modeled by open lists of the form

```
Z0 = [F1,...,Fm | Z]
```

(encoding the state description $\texttt{Z0} = \texttt{F1} \circ \ldots \circ \texttt{Fm} \circ \texttt{Z}$), along with constraints

```
not_holds(F, Z)
not_holds_all([X1,...,Xk], F, Z)
```

encoding, resp., the negative statements $(\exists \boldsymbol{y})\, \neg \mathit{Holds}(\texttt{F}, \texttt{Z})$ (where $\boldsymbol{y}$ are the variables occurring in F) and $(\exists \boldsymbol{y})(\forall \texttt{X1}, \ldots, \texttt{Xk})\, \neg \mathit{Holds}(\texttt{F}, \texttt{Z})$ (where $\boldsymbol{y}$ are the variables occurring in F except $\texttt{X1}, \ldots, \texttt{Xk}$). These two constraints are used to bypass the problem of 'negation-as-failure' with incomplete states. In order to process these constraints, so-called declarative Constraint Handling Rules [5] have been defined and proved correct under the foundational axioms of the Fluent Calculus. In addition, the core of FLUX contains definitions for `holds(F,Z)`, by which is encoded macro (1), and `update(Z1,ThetaP,ThetaN,Z2)`, which encodes the state equation $\texttt{Z2} = (\texttt{Z1} \circ \texttt{ThetaP}) - \texttt{ThetaN}$. The following, for instance, is the FLUX encoding of our state update axioms (4) (ignoring preconditions) and the initial specification (2):

```
state_update(Z1, take(X), Z2) :-
  update(Z1, [carries(X)], [on_table(X)], Z2).

state_update(Z1, test(X,Y), Z2) :-
  holds(acidic(X), Z1), update(Z1, [red(Y)], [], Z2) ;
  not_holds(acidic(X), Z1), update(Z1, [], [], Z2).

init(Z0) :-
   holds(on_table(a), Z0),
   holds(on_table(b), Z0), holds(on_table(c), Z0),
   not_holds(red(p), Z0), not_holds_all([X], carries(X), Z0),
   (not_holds(acidic(b), Z0) ; not_holds(acidic(c), Z0)),
   duplicate_free(Z0).
```

where the constraint `duplicate_free(Z)` means that list `Z` does not contain multiple occurrences. Suppose, e.g., that Litmus paper P is red after testing solution B, then it follows that B must have been acidic but not C:

```
?- init(Z0), state_update(Z0, test(b,p), Z1), holds(red(p), Z1).

Z0 = [on_table(a),on_table(b),on_table(c),acidic(b) | _Z]
Constraints:
not_holds(acidic(a), _Z)
...
```

2.3 Knowledge Update Axioms

To represent knowledge in the Fluent Calculus and to reason about sensing actions, the predicate $\mathit{KState} : \text{SIT} \times \text{STATE}$ has been introduced in [23]. An instance $\mathit{KState}(s, z)$ means that according to the knowledge of the planning robot, z is a possible state in situation s. A fluent is then known to hold (resp.

not hold) in a situation just in case it is true (resp. false) in all possible states; and it is known whether a fluent holds just in case it is known to hold or known not to hold:

$$\begin{aligned} Knows(f,s) &\stackrel{\text{def}}{=} (\forall z)\,(KState(s,z) \supset Holds(f,z)) \\ Knows(\neg f,s) &\stackrel{\text{def}}{=} (\forall z)\,(KState(s,z) \supset \neg Holds(f,z)) \\ Kwhether(f,s) &\stackrel{\text{def}}{=} Knows(f,s) \vee Knows(\neg f,s) \end{aligned} \tag{5}$$

These macros generalize to the knowledge of arbitrary non-atomic formulas in a natural way. A foundational axiom stipulates correctness of state knowledge:

$$KState(s, State(s))$$

The Frame Problem for knowledge is solved by axioms that determine the relation between the possible states before and after an action. More formally, the effect of an action $A(\boldsymbol{x})$, be it sensing or not, on the knowledge is specified by a *knowledge update axiom*,

$$\begin{aligned} &Poss(A(\boldsymbol{x}),s) \supset \\ &\quad (\forall z)\,(KState(Do(A(\boldsymbol{x}),s),z) \equiv (\exists z')(KState(s,z') \wedge \Psi(z,z',s)))\,) \end{aligned} \tag{6}$$

In case of non-sensing actions, formula Ψ defines what the robot knows of the effects of the action. In case of sensing actions, formula Ψ restricts the possible states in such a way that the sensed property becomes known. In particular, let the generic ACTION term $Sense(f)$ denote sensing whether a fluent f holds, then:

$$\begin{aligned} &Poss(Sense(f),s) \supset \\ &\quad KState(Do(Sense(f),s),z) \equiv \\ &\qquad KState(s,z) \wedge [Holds(f,z) \equiv Holds(f,s)] \end{aligned} \tag{7}$$

That is to say, among the states possible in s only those are still possible after sensing which agree with the actual state of the world as far as the sensed fluent is concerned. A crucial immediate consequence is that sensing always causes the truth value of a property to be known [23].[3]

Based on knowledge update axioms, the inferential Frame Problem for knowledge is solved with the help of a simple inference schema. Suppose given an axiom which summarizes *all* that is known of a situation s, that is, $KState(s,z) \equiv \Phi(z)$. Suppose further, for the sake of argument, that $Poss(A(\boldsymbol{x}),s)$, then (6) entails

$$KState(Do(A(\boldsymbol{x}),s),z) \equiv (\exists z')\,(\Phi(z') \wedge \Psi(z,z',s)) \tag{8}$$

which provides a specification of what is known in the successor situation.

In accordance with the classical notion of planning by deduction, conditional plans in the Fluent Calculus are first-order citizens, composed of the primitive

[3] While for the sake of abstraction axiom (7) specifies an ideal sensor for qualitative fluents, nondeterministic knowledge update axioms can be used to model sensor noise when sensing quantitative fluents.

actions of a domain and using the standard functions ϵ (empty action), $a_1; a_2$ (sequential composition), and $If(f, a_1, a_2)$ (conditional branching). Preconditions, state update, and knowledge update for these ACTION functions are defined by foundational axioms [23]. Here is an example of a situation representing a conditional plan:

$$\begin{aligned} S \;=\; Do(\, & If(Red(P), Take(C), Take(B)), \\ & Do(Sense(Red(P)), Do(Test(B, P), S_0))) \end{aligned} \tag{9}$$

Applied to our example initial specification, (2), this plan can be proved to achieve the goal of getting a chemical solution which is known not to be acidic:

$$(\exists x)\,(Knows(Carries(x), S) \wedge Knows(\neg Acidic(x), S))$$

3 Identifying Knowledge with Incomplete States

While the explicit notion of possible states leads to an extensive framework for reasoning about knowledge and sensing, automated deduction becomes considerably more intricate by the introduction of the modality-like *KState* predicate. In this section, we develop the foundations for an inference method which avoids separate update of knowledge and states. To this end, we show how knowledge updates are implicitly obtained by progressing an incomplete state through state update axioms.

Our approach rests on two assumptions. First, the planning robot needs to know the given initial specification $\Phi(State(S_0))$, and this is all it knows of S_0, that is, $KState(S_0, z) \equiv \Phi(z)$. Second, the robot must have accurate knowledge of its own actions:

Definition 1. *A set of axioms* Σ represents accurate effect knowledge *if for each non-sensing* ACTION *function* A, Σ *contains a unique state update axiom*

$$Poss(A(\boldsymbol{x}), s) \supset \Gamma_A\{z/State(Do(A(\boldsymbol{x}), s)), z'/State(s))\} \tag{10}$$

(where $\Gamma_A(\boldsymbol{x}, z, z')$ *is a first-order formula with free variables among* $\boldsymbol{x}, z, z'$ *and without a sub-term of sort* SIT*) and a unique knowledge update axiom which is equivalent to*

$$\begin{aligned} & Poss(A(\boldsymbol{x}), s) \supset \\ & \quad (\forall z)\,(KState(Do(A(\boldsymbol{x}), s), z) \equiv \\ & \qquad\qquad (\exists z')(KState(s, z') \wedge \Gamma_A(\boldsymbol{x}, z, z'))\,) \end{aligned} \tag{11}$$

Put in words, the possible states after a non-sensing action are those which would be the result of actually performing the action in one of the previously possible states.

Accurate knowledge of effects suffices to ensure that the possible states after a non-sensing action can be obtained by progressing a given state specification

through the state update axiom for that action. The effect of sensing, on the other hand, cannot be obtained in the same fashion. To see why, let S be a situation and consider the knowledge specification

$$KState(S, z) \equiv [Holds(Red(P), z) \equiv Holds(Acidic(A), z)] \tag{12}$$

(which may have been inferred as the result of a $Test(A, P)$ action). Suppose that $Poss(Sense(Red(P)), S)$, then the knowledge update axiom (7) for $Sense(Red(P))$ yields two models for $KState(Do(Sense(Red(P)), S), z)$, the first of which satisfies

$$KState(Do(Sense(Red(P)), S), z) \equiv \\ Holds(Red(P), z) \wedge Holds(Acidic(A), z)$$

(for all z) whereas the other one satisfies

$$KState(Do(Sense(Red(P)), S), z) \equiv \\ \neg Holds(Red(P), z) \wedge \neg Holds(Acidic(A), z)$$

(again for all z). The first model represents the case where $Red(P)$ actually holds in S while the second model represents the case where $Red(P)$ actually does not hold in S. Due to the existence of these two models there can be no unique specification of the form $KState(Do(Sense(Red(P)), S), z) \equiv \Phi(z)$ entailed by (12) and (7). Hence, the effect of a sensing action cannot be obtained by straightforward progression.

In order to account for different models for $KState$ caused by sensing, we introduce the notion of a *sensing history* ς as a finite, possibly empty list of 0's and 1's. A history is meant to describe the outcome of each sensing action in a sequence of actions. For the sake of simplicity, we assume that the only sensing action is the generic $Sense(f)$ with knowledge update axiom (7) and state update axiom $State(Do(Sense(f), s)) = State(s)$.

For the formal definition of progression we also need the notion of an *action sequence* σ as a finite, possibly empty list of ground ACTION terms. An action sequence corresponds naturally to a situation, which we denote by S_σ:

$$S_{[\,]} \stackrel{\text{def}}{=} S_0 \quad \text{and} \quad S_{[A(\,\boldsymbol{t}\,)\,|\,\sigma]} \stackrel{\text{def}}{=} Do(A(\,\boldsymbol{t}\,), S_\sigma)$$

We are now in a position to define, inductively, a *progression operator* $\mathcal{P}(\sigma, \varsigma, z)$, by which an initial state specification $\Phi(State(S_0))$ is progressed through an action sequence σ wrt. a sensing history ς, resulting in a formula specifying z:

$$\mathcal{P}([\,], \varsigma, z) \stackrel{\text{def}}{=} \Phi(z) \quad \text{if } \varsigma = [\,] \tag{13}$$

$$\mathcal{P}([A(\boldsymbol{t}) \,|\, \sigma], \varsigma, z) \stackrel{\text{def}}{=} (\exists z')\,(\mathcal{P}(\sigma, \varsigma, z') \wedge \Gamma_A(\boldsymbol{t}, z, z')) \\ \text{if } A \text{ non-sensing with state update (10)} \tag{14}$$

$$\mathcal{P}([Sense(f) \,|\, \sigma], \varsigma, z) \stackrel{\text{def}}{=} \begin{array}{ll} \mathcal{P}(\sigma, \varsigma', z) \wedge \neg Holds(f, S_\sigma) & \text{if } \varsigma = [0 \,|\, \varsigma'] \\ \mathcal{P}(\sigma, \varsigma', z) \wedge Holds(f, S_\sigma) & \text{if } \varsigma = [1 \,|\, \varsigma'] \end{array} \tag{15}$$

In case the length of the history ς does not equal the number of sensing actions in σ, we define $\mathcal{P}(\sigma, \varsigma, z)$ as *False*. As the main result, progression provides a provably correct inference method for knowledge update.[4]

Theorem 2. *Consider the initial state and knowledge* $\Sigma_0 = \{\Phi(State(S_0)), KState(S_0, z) \equiv \Phi(z)\}$ *and let* Σ *be the foundational axioms plus a set of domain axioms representing accurate effect knowledge. Let* σ *be an action sequence such that* $\Sigma \cup \Sigma_0 \models POSS(\sigma)$. *Then for any model* $\mathcal{M}$ *of* $\Sigma_0 \cup \Sigma$ *and any valuation* ν,

$$\mathcal{M}, \nu \models KState(S_\sigma, z) \quad \textit{iff} \quad \mathcal{M}, \nu \models \mathcal{P}(\sigma, \varsigma, z) \text{ for some } \varsigma$$

Proof (sketch) The proof is by induction on σ. The base case $\sigma = [\,]$ follows by (13) and Σ_0. The induction step for $\sigma = [A(\boldsymbol{t}) \,|\, \sigma']$ with $A(\boldsymbol{t})$ being a non-sensing action follows by (14) and knowledge update axiom (11). The induction step for $\sigma = [Sense(f) \,|\, \sigma']$ follows by (15) and knowledge update axiom (7).

This theorem serves as the formal justification for the FLUX encoding of knowledge and sensing. The generic sensing action $Sense(f)$ is encoded by a state update axiom which carries as additional argument the result of sensing, where the sensing value is either 0 or 1:

```
state_update(Z, sense(F), Z, SV) :-
   not_holds(F, Z), SV=0 ; holds(F, Z), SV=1.
```

The definition of progression is a direct encoding of (13)–(15):

```
p([], [], Z) :- init(Z).
p([A|S], H2, Z2) :- p(S, H1, Z1),
                    ( state_update(Z1, A, Z2), H2=H1 ;
                      state_update(Z1, A, Z2, SV), H2=[SV|H1] ).
```

The FLUX definitions for $Knows(f, s)$, $Knows(\neg f, s)$, and $Kwhether(f, s)$ then follow from Theorem 2.

Corollary 3. *Let* ϕ *be a* FLUENT *term. Under the assumptions of Theorem 2,*

1. $\Sigma_0 \cup \Sigma \models Knows(\phi, S_\sigma)$ *iff there is no model* $\mathcal{M}$ *of* $\Sigma_0 \cup \Sigma$, *no valuation* ν, *and no history* ς *such that* $\mathcal{M}, \nu \models \mathcal{P}(\sigma, \varsigma, z) \wedge \neg Holds(\phi, z)$.
2. $\Sigma_0 \cup \Sigma \models Knows(\neg\phi, S_\sigma)$ *iff there is no model* $\mathcal{M}$ *of* $\Sigma_0 \cup \Sigma$, *no valuation* ν, *and no history* ς *such that* $\mathcal{M}, \nu \models \mathcal{P}(\sigma, \varsigma, z) \wedge Holds(\phi, z)$.
3. $\Sigma_0 \cup \Sigma \models Kwhether(\phi, S_\sigma)$ *iff there is no model* $\mathcal{M}$ *of* $\Sigma_0 \cup \Sigma$, *no valuation* ν, *and no history* ς *such that* $\mathcal{M}, \nu \models \mathcal{P}(\sigma, \varsigma, z_1) \wedge Holds(\phi, z_1) \wedge \mathcal{P}(\sigma, \varsigma, z_2) \wedge \neg Holds(\phi, z_2)$.

Proof (sketch) Follows from Theorem 2 and macro (5).

Hence:

[4] Below, $POSS(\sigma)$ means that σ is possible in S_0, that is, $POSS([\,]) \stackrel{\text{def}}{=} True$ and $POSS([A(\boldsymbol{t}) \,|\, \sigma]) \stackrel{\text{def}}{=} POSS(\sigma) \wedge Poss(A(\boldsymbol{t}), S_\sigma)$.

```
knows(F, S) :- is_fluent(F), \+ ( p(S, _, Z), not_holds(F, Z) ).
knows(-(F), S) :- is_fluent(F), \+ ( p(S, _, Z), holds(F, Z) ).
kwhether(F, S) :- is_fluent(F), \+ ( p(S, H, Z1), holds(F, Z1),
                  p(S, H, Z2), not_holds(F, Z2) ).
```

where `is_fluent` shall be true if the argument constitutes a FLUENT term of the language. Recall, for instance, the example initial state of Section 2. Whether solution A is acidic is still unknown after testing it but will be known after further sensing the color of the Litmus strip—though it cannot be predicted that it is acidic (nor, of course, that it is not):

```
?- \+ kwhether(acidic(a), [test(a,p)]),
   kwhether(acidic(a), [sense(red(p)),test(a,p)]),
   \+ knows(acidic(a), [sense(red(p)),test(a,p)]).
yes
```

The range of Theorem 2 includes nondeterministic actions. The latter may in particular cause loss of knowledge [17]; e.g.,

```
state_update(Z1, dilute(X), Z2) :-
  Z2 = Z1 ; update(Z1, [], [acidic(X)], Z2).

?- kwhether(acidic(a), [dilute(a),sense(red(p)),test(a,p)]).
no
```

4 Conditional Plans and Plan Skeletons

The reified conditional plans of the Fluent Calculus are encoded in FLUX as possibly nested lists of actions, in the order of execution; e.g.,

```
[test(b,p), sense(red(p)), if(red(p),[take(c)],[take(b)])]
```

represents conditional plan (9) from above. A planning problem with incomplete states and sensing actions is the problem of finding a conditional plan which can be proved to be executable and to achieve the goal under any circumstances. Therefore, if a conditional action is inserted into a plan, then each branch must be searched individually. To this end, we introduce the auxiliary actions $Commit(f)$ and $Commit(\neg f)$, which, formally, do not affect the world state but the knowledge:

$$KState(Do(Commit(f), s), z) \equiv KState(s, z) \wedge Holds(f, z)$$
$$KState(Do(Commit(\neg f), s), z) \equiv KState(s, z) \wedge \neg Holds(f, z)$$

In terms of FLUX:

```
state_update(Z, commit(F), Z) :- \+ F = -(_), holds(F, Z).
state_update(Z, commit(-(F)), Z) :- not_holds(F, Z).
```

In principle, the FLUX clauses we arrived at can readily be used by a simple forward-chaining search algorithm. Enumerating the set of plans, including all possible sensing actions, a solution will eventually be found if only the problem is solvable. However, planning with incomplete states usually involves a considerable search space, and the possibility to generate conditional plans only enlarges it. The concept of nondeterministic robot programs has been introduced in GOLOG as a powerful heuristics for planning, where only those plans are searched which match a given skeleton [15]. This avoids considering obviously useless actions such as ineffectual sensing. In the following, we generalize this concept to incomplete states and state knowledge. Our major extension concerns conditionals, which we resolve at planning time only if the state knowledge suffices to do so; otherwise, a branching point is introduced, leading to a conditional plan by need.

Similar to GOLOG we use a macro $do(\delta, \sigma, p)$ where δ is a robot program (represented as sequence of commands), σ a sequence of actions (possibly including the auxiliary *Commit*), and p is a (possibly conditional) plan. The intended reading is that executing δ in situation S_σ may result in the executable plan p. The crucial extension to GOLOG is this new definition of a conditional:

$$\begin{aligned} do(\texttt{if}\ f\ \texttt{then}\ \delta_1\ \texttt{else}\ \delta_2, \sigma, p) \stackrel{\text{def}}{=} (\exists p_1, p_2)\,(\,&Kwhether(f, S_\sigma) \wedge \\ &do(\delta_1, [Commit(f) \mid \sigma], p_1) \wedge \\ &do(\delta_2, [Commit(\neg f) \mid \sigma], p_2) \wedge \\ &p = [If(f, p_1, p_2)] \end{aligned} \tag{16}$$

The other standard macros of GOLOG are straightforwardly adapted to the Fluent Calculus. We just mention those which will be used in our example below, namely, primitive actions, testing, and nondeterministic choice of sub-programs (denoted by $\delta_1 \# \delta_2$) and of arguments (denoted by $(\pi x)\delta$).

$$do([\,], \sigma, p) \stackrel{\text{def}}{=} p = [\,]$$

$$\begin{aligned} do([a \mid \delta], \sigma, p) \stackrel{\text{def}}{=} (\exists p')\,(\,&Poss(a, S_\sigma) \wedge \\ &do(\delta, [a \mid \sigma], p') \wedge p = [a \mid p']\,) \end{aligned}$$

$$do([\{\neg\}Knows(f)? \mid \delta], \sigma, p) \stackrel{\text{def}}{=} Knows(\{\neg\}f, \sigma) \wedge do(\delta, \sigma, p)$$

$$do([\{\neg\}Kwhether(f)? \mid \delta], \sigma, p) \stackrel{\text{def}}{=} \{\neg\}Kwhether(f, \sigma) \wedge do(\delta, \sigma, p)$$

$$do([\delta_1 \# \delta_2 \mid \delta], \sigma, p) \stackrel{\text{def}}{=} do(\delta_1 + \delta, \sigma, p) \vee do(\delta_1 + \delta, \sigma, p)$$

$$do((\pi x)\delta, \sigma, p) \stackrel{\text{def}}{=} (\exists x)\, do(\delta, \sigma, p)$$

where $\delta + \delta'$ denotes concatenation of two programs. The encoding in FLUX is straightforward; we just mention the clause which encodes the conditional:

```
do([if(F,E1,E2)|L], S, P) :-
  is_fluent(F),
  append(E1, L, L1), append(E2, L, L2),
```

```
( knows(F, S), !, do(L1, S, P) ;
  knows(-(F), S), !, do(L2, S, P) ;
  kwhether(F, S), do(L1, [commit(F)|S], P1),
                  do(L2, [commit(-(F))|S], P2),
                  P = [if(F,P1,P2)] ).
```

That is to say, if the condition can be decided in advance, then the corresponding branch is chosen; otherwise, a conditional plan is generated and both branches are searched. Regarding the latter case, notice that it is checked (using `kwhether`) that it will be possible to evaluate the condition at execution time (c.f. (16)); if not, then the clause fails as the resulting plan would not be executable.

The empty robot program terminates successfully with the empty plan if the planning goal is satisfied:

```
do([], S, []) :- goal(S).
```

As an example, consider the following recursive robot program, which can be used to find among any selection of chemical solutions a non-acidic one with a sufficient supply of Litmus paper:

```
proc(find_non_acidic, [pi(x,[(knows(on_table(x)))?,
                             (not knows(acidic(x)))?,
                             []#[test_acidity(x)],
                             if(acidic(x), [find_non_acidic],
                                           [take(x)])])]).

proc(test_acidity(X), [(not kwhether(acidic(X)))?,
                       pi(y,[test(X,y),sense(red(y))])]).
```

Put in words, to find a non-acidic solution, pick one which is not known to be acidic. It may be necessary to test the solution. (The first item in the body of the auxiliary procedure `test_acidity` avoids redundant testing.) If the selected solution is acidic, try to find another one, else grab it.

Consider, now, the goal to get a non-acidic solution,

```
goal(S) :- knows(carries(X), S), knows(-(acidic(X)), S).
```

With suitable domain clauses defining `is_fluent` and the action preconditions, the program will generate the following plan given the example initial specification of Section 2:

```
?- do([find_non_acidic], [], P)

P = [test(b,p), sense(red(p)), if(red(p),[take(c)],[take(b)])]
```

The reader may notice that it suffices to test solution B; if it turns out to be acidic, then C must be non-acidic.[5] It is worth stressing that even with the

[5] A second solution to the planning problem is of course to test solution C and to branch upon the result accordingly.

given plan skeleton, it is necessary to find the right sensing action. In particular, the system has to backtrack over the attempt to test solution A (which renders unusable the only available Litmus paper)!

5 Related Work

A distinguishing feature of our system is its expressiveness in comparison to most existing systems for planning with knowledge and sensing. In [9] an implementation is described for which a semantics is given based on the general Situation Calculus solution to the Frame Problem for knowledge of [20]. However, the implementation is based on the notion of an incomplete state as a triple of true, false, and unknown propositional fluents. The same representation is used in the logic programming systems [2,16], which are both given semantics by a three-valued variant [2] of the Action Description Language [6]. This restricted notion of incomplete states does not allow for handling any kind of *disjunctive* information. As a consequence, none of the aforementioned systems can solve planning problems that require to derive implicit knowledge (as in the Litmus scenario) or reasoning by cases. The latter is necessary whenever an action has conditional effects depending on whether some unknown fluent is true or false, but where both conditional effects suffice to achieve the goal [2]. Similar restrictions apply to the approach of [7], based on Description Logic.

The only existing systems with a general solution to the Frame Problem for knowledge is [19]. However, this Prolog implementation cannot be used for planning with sensing as it does not allow to *search* for suitable sensing actions. Rather, the user is supposed to provide GOLOG programs where all necessary sensing actions have been correctly planned. Likewise restricted to plan verification is the approach [10], which is based on a special epistemic propositional logic. In contrast, our system is designed for solving planning problems as it allows to backtrack over sensing actions that lead to a dead end.

The semantics of our logic program is given by previous work on integrating a solution to the Frame Problem for knowledge into the Fluent Calculus [23]. This axiomatization technique is related to the Situation Calculus-based formalization of [20]. The basic idea there is to represent state knowledge by a binary situation-situation relation $K(s, s')$, meaning that as far as the robot knows in situation s it could as well be in situation s'. Hence, every given fact about any such s' is considered possible by the robot. Having readily available the explicit notion of a state in the Fluent Calculus, our formalization avoids this indirect encoding of state knowledge, which is intuitively less appealing because it seems that a robot should always know exactly which *situation* it is in—after all, situations in the Situation Calculus are merely sequences of actions that have been or will be taken by the robot [13]. In view of the computational challenge raised by the Frame Problem for knowledge, a crucial advantage of our approach is also the simple inference scheme (8) provided by the concept of knowledge update axioms.

A conceptually different semantical approach has been proposed in [17] as an extension of the Action Description Language which is more powerful than the abovementioned [2]. Incomplete knowledge is formalized by a so-called epistemic state, which is a set of possible sets of possible states. Intuitively, an epistemic state corresponds to the set of models for our *KState* predicate. We therefore suspect that our logic program can be shown to provide a sound and complete proof procedure for this semantics, too, but the formal details have yet to be worked out.

6 Discussion

We have developed the formal foundations for an effective inference method for state knowledge in the presence of incomplete states, nondeterministic actions, and sensing. Conditional plans are computed by reasoning about knowledge based on progression and with the help of a generalized concept of nondeterministic robot programs as search heuristics. The resulting extension of the high-level programming language FLUX exhibits a clear distinction between nondeterministic actions and nondeterminism in the heuristics. The latter needs to be resolved at planning time, possibly by introducing a branching point into a plan. Nondeterminism in state update axioms, on the other hand, is respected when verifying knowledge preconditions or proving that the plan is correct under any outcome.

We have successfully applied FLUX to the high-level control of a simple Lego robot [12] as well as a Pioneer-2, both of which perform delivery tasks and need to generate conditional plans which include sensing whether doors are closed.

Future work will be to extend the progression operator to actions with ramifications and to concurrency, in order to provide the formal justification for inferring knowledge in more complex domains. Furthermore, off-line planning in FLUX should be interleaved with on-line execution of sensing actions, following the argument of [8] that pure off-line planning can often be inefficient in the presence of sensing. Finally, while sensor noise has been ignored in all our applications thus far, the concept of knowledge update axioms can be readily applied to model nondeterministic outcomes, and hence to account for noise [1]. We currently pursue the axiomatization and implementation of sensor noise within our approach along this line.

References

1. F. Bacchus, J. Halpern, and H. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artif. Intell.*, 111(1–2):171–208, 1999.
2. C. Baral and T. C. Son. Approximate reasoning about actions in presence of sensing and incomplete information. In J. Maluszynski, ed., *Proc. of ILPS*, p. 387–401, Port Jefferson, 1997.
3. W. Bibel. Let's plan it deductively! *Artif. Intell.*, 103(1–2):183–208, 1998.
4. R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2:189–208, 1971.

5. T. Frühwirth. Theory and practice of constraint handling rules. *J. of Logic Programming*, 37(1–3):95–138, 1998.
6. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *J. of Logic Programming*, 17:301–321, 1993.
7. G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing for a mobile robot. In *Proc. of the European Conf. on Planning*, vol. 1348 of *LNAI*, p. 158–170. Springer, 1997.
8. G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, ed.'s, *Logical Foundations for Cognitive Agents*, p. 86–102. Springer, 1999.
9. K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. In L. C. Aiello, J. Doyle, and S. Shapiro, ed.'s, *Proc. of KR*, p. 174–185, Cambridge, 1996.
10. A. Herzig, J. Lang, D. Longin, and T. Polascek. A logic for planning under partial observability. In H. Kautz and B. Porter, ed.'s, *Proc. of AAAI*, p. 768–773, 2000.
11. G. Lakemeyer. On sensing and off-line interpreting GOLOG. In H. Levesque and F. Pirri, ed.'s, *Logical Foundations for Cognitive Agents*, p. 173–189. Springer, 1999.
12. H. Levesque and M. Pagnucco. Legolog: Inexpensive experiments in cognitive robotics. In *Cognitive Robotics Workshop at ECAI*, p. 104–109, Berlin, 2000.
13. H. Levesque, F. Pirri, and R. Reiter. Foundations for a calculus of situations. *Electronic Transactions on Artif. Intell.*, 3((1–2)):159–178, 1998. `http://www.ep.liu.se/ea/cis/1998/018/`.
14. H. Levesque. What is planning in the presence of sensing? In B. Clancey and D. Weld, ed.'s, *Proc. of AAAI*, p. 1139–1146, Portland, 1996.
15. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *J. of Logic Programming*, 31(1–3):59–83, 1997.
16. J. Lobo. COPLAS: A conditional planner with sensing actions. In *Cognitive Robotics*, vol. FS–98–02 of *AAAI Fall Symposia*, p. 109–116. AAAI Press 1998.
17. J. Lobo, G. Mendez, and S. Taylor. Adding knowledge to the action description language $\mathcal{A}$. In B. Kuipers and B. Webber, ed.'s, *Proc. of AAAI*, p. 454–459, Providence, 1997.
18. R. Moore. A formal theory of knowledge and action. In J. R. Hobbs and R. C. Moore, ed.'s, *Formal Theories of the Commonsense World*, p. 319–358. Ablex, 1985.
19. R. Reiter. On knowledge-based programming with sensing in the situation calculus. In *Cognitive Robotics Workshop at ECAI*, p. 55–61, Berlin, 2000.
20. R. Scherl and H. Levesque. The frame problem and knowledge-producing actions. In *Proc. of AAAI*, p. 689–695, Washington, 1993.
21. M. Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artif. Intell.*, 111(1–2):277–299, 1999.
22. M. Thielscher. The fluent calculus: A specification language for robots with sensors in nondeterministic, concurrent, and ramifying environments. Technical Report CL-2000-01, Dresden University of Technology, 2000. `http://www.cl.inf.tu-dresden.de/~mit/publications/reports/CL-2000-01.pdf`
23. M. Thielscher. Representing the knowledge of a robot. In A. Cohn, F. Giunchiglia, and B. Selman, ed.'s, *Proc. of KR*, p. 109–120, Breckenridge, 2000.

Multi-agent Systems as Intelligent Virtual Environments

George Anastassakis[2], Tim Ritchings[1], and Themis Panayiotopoulos[2]

[1] School of Sciences, University of Salford,
Newton Building, University of Salford, Salford, M5 4WT
T.Ritchings@salford.ac.uk
[2] Knowledge Engineering Lab, Department of Informatics,
University of Piraeus, Piraeus, 185 34, Greece
{anastas, themisp}@unipi.gr

Abstract. Intelligent agent systems have been the subject of intensive research over the past few years; they comprise one of the most promising computing approaches ever, able to address issues that require abstract modelling and higher level reasoning. Virtual environments, on the other hand, offer the ideal means to produce simulations of the real world for purposes of entertainment, education, and others. The merging of these two fields seems to have a lot to offer to both research and applications, if progress is made on a co-ordinated manner and towards standardization. This paper is a presentation of VITAL, an intelligent multi-agent system able to support general-purpose intelligent virtual environment applications.

1 Introduction

Probably one of the most exciting and promising scientific fields ever, the field of *intelligent agents* is still the subject of major controversy over its origin, formal background, definitions, methods, applications and future directions. The notion of an intelligent agent, indisputably challenging to define precisely, has been used to characterize a vast number of approaches and applications, ranging from simple softbots to complex, large-scale industrial control systems.

Recent attempts to merge intelligent agent approaches with virtual reality and artificial life have given birth to the field of *intelligent virtual environments (IVEs)*. An IVE is a virtual environment resembling the real world (or similar), inhabited by autonomous intelligent entities exhibiting a variety of behaviours. These entities may be simple static or dynamic objects (a revolving sun, traffic lights, etc.), virtual representations of life forms (virtual animals and humans), avatars of real-world users entering the system, and others. In fact, the structure and contents of a virtual environment are only restricted by the nature of the target application and the designer's imagination - and, of course, the amount of computing power available.

Today, IVEs are employed in a variety of areas, mainly relating to simulation, entertainment, and education. Sophisticated simulated environments of different types (open urban spaces, building interiors, streets, etc) can significantly aid in architectural design, civil engineering, traffic and crowd control, and others. In

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 381–395, 2001.

addition, precisely modelled simulations of real-world equipment (vehicles, aircrafts, etc) not only can be tested at reduced cost and risk, but also more accurate results can be obtained thanks to the additional element of control by and interaction with intelligent, thus closer to real life, entities. Moreover, IVEs have set new standards in computer-aided entertainment, through outstanding examples of computer games involving large, life-like virtual worlds (where imaginative scenarios are to be challenged), interactive drama (where the user is an active participant in the plot) virtual story-telling, and many other areas where immersion and believability are key factors. Concluding, IVE-based educational systems incorporate believable tutoring characters and sophisticated data representation techniques, resulting in the stimulation of user interest and perceptual ability, thus providing a novel, effective and enjoyable learning experience.

Despite the fact that an intelligent agent is the ideal metaphor for representing intelligent inhabitants inside an IVE, surprisingly little effort has been directed towards a formal and co-ordinated merging of intelligent agent systems and virtual reality techniques to produce IVEs fully exploiting the advantages of both fields. This paper is a presentation of our attempt to contribute to such an initiative: a fully functional intelligent agent system with the ability to support virtual intelligent agents embodied inside simulated worlds represented using VR techniques. Along with a number of other features of both practical and scientific value, the system can be used for a variety of purposes, acting as either an IVE for one of the application areas mentioned above, or a typical multi-agent system employed in classical application domains, such as control systems, distributed problem solving, resource allocation and many others. In the rest of this paper, a discussion of relevant research work is given in section two. Section three is a thorough presentation of the proposed system, while section four is a layout of additional work carried out towards multi-agent support. An example of the system's operation is given in section five.

2 Related Work

The Beliefs-Desires-Intentions (BDI) model [4] is probably the most popular approach towards the design of intelligent agents, mainly due to its ability to trigger behaviours driven by conceptually modelled intentions and goals rather than explicit procedural information. In addition, it seems to be a functional abstraction for the higher-level reasoning processes of the human mind, those that are related to action selection and the focusing of intelligent reasoning processes on specific desired states. The BDI model has been adopted in a significant number of implementations:

In [3], Bratman et al. present the Intelligent Resource-bounded Machine Architecture (IRMA), an architecture for resource-bounded (mainly in terms of computational power) deliberative agents, based on the BDI model. IRMA agents consist of four main modules: a means-end planner, an opportunity analyser, a filtering process and a deliberation procedure. In addition, they contain a plan library, and data structures to store beliefs, desires and intentions.

Jennings in [10] proposes GRATE, an architecture clearly focused on co-operative problem solving through agent collaboration. Central to the entire architecture is the notion of joint-intentions. In fact, even though GRATE is a deliberative architecture based on the BDI model, it is specifically referred to as a belief-desire-joint-intention architecture.

The BDI model has provided valuable theoretical grounds upon which the development of several other architectures and approaches, such as hybrid and layered agents, was based [9]:

The Procedural Reasoning System (PRS) [7] is a hybrid system, where beliefs are expressed in first-order predicate logic and desires represent system behaviours instead of fixed goals. PRS includes a plan library containing a set of partial plans, called knowledge areas, each associated with an invocation condition. Knowledge areas might be executed due to goal-driven reasoning or as a response to sensory data; this way, the agent is capable for both deliberative and reactive behaviours.

Muller in [12] proposes INTERRAP, a layered agent architecture focusing on the requirements of situated and goal-directed behaviour, efficiency and co-ordination. INTERRAP agents consist of a world interface, a behaviour-based, a plan-based and a co-operation component, each affecting agent behaviours at a different level of social and functional abstraction..

In [16], Sycara et al. present the Reusable Task Structure-based Intelligent Network Agents (RETSINA) architecture. The architecture consists of three types of agents: interface, task agents and, information agents.

Due to its apparent focus on high-level reasoning and generation of elaborate behavioural patterns, the BDI model seems to be inadequate to efficiently and effectively model all aspects of intelligent reasoning. However, any system that needs to exhibit goal-driven behaviour should incorporate, among others, a BDI-based or equivalent component.

The merging of intelligent agent systems, artificial life and classical VR techniques has given birth to the field of *Intelligent Virtual Environments (IVEs)*. Typical examples involving IVEs and general virtual agents include Humanoid [2], Creatures [8], Artificial Fishes [17], and others.

The CoMMA-COGs [5] project (Cooperative Man Machine Architectures - Cognitive Architecture for Social Agents) is an architecture for Multi-Agent systems and animated virtual environments, developed by the German Research Center for Artificial Intelligence. The system employs traditional multi-agent research approaches. Furthermore, it supports self-organization of agent societies, so that external users perceive them as units, and, thus, being unaware of the underlying organization processes. In addition, resource-awareness allows agents to perform in unpredictable environments while flexibly managing their resources. In general, IVEs tend to focus on either the virtual representation and embodiment side, or the intelligence side. Full benefit has not yet been taken of the combined advantages of intelligent multi-agent systems and virtual environments. A complex, accurately modelled and general-purpose IVE, inhabited by numerous believable entities driven by strong and effective AI reasoning processes, is yet to be presented.

A predecessor to the VITAL system and a first effort towards an intelligent agent system architecture with the ability to support IVE applications, the DIVA architecture, developed by the Knowledge Engineering Lab of the University of Piraeus, was presented in [18].

3 Overview of the VITAL System

The system presented in this paper is called *VITAL*, an acronym for *Virtual InTelligent Agents with Logic*. It represents an attempt to explore the world of

intelligent agent systems and intelligent virtual environments, initiated in UMIST, UK [1]. VITAL is a simulation of a maze world. Agents are required to explore a given maze, locate specific items inside it and process them in response to user instructions, e.g. move them to given locations. Agents have no initial knowledge of either the structure of the maze or of the items' locations. The system is monitored using a 3D viewer supporting free world navigation and several configuration options.

The system was designed and implemented with the following goals and requirements in mind:

1. *Wide range of applications:* the system should support a wide range of areas.
2. *Agent effectiveness, independence and agility:* for a given area, agents should be effective, that is, achieving what they have been assigned; furthermore, they should require no user intervention while doing so; finally, they should be agile, maintaining their effectiveness even when the environment changes unpredictably within the limits of a defined domain.
3. *Distributed and modular structure:* the system should be distributed, consisting of discrete co-operating components executing on possibly different machines across a network; in addition, modularity should allow the insertion and removal of components at runtime without interrupting the system's operation.
4. *Significant research potential:* the system should be designed so that scientific experimentation is inherently.
5. *Intuitive observation and monitoring capabilities*.
6. *Sophisticated implementation:* the system should be implemented using modern, specialized software tools and latest technologies, essentially comprising a high-quality software product.
7. *Extendibility and reusability:* the system should be highly extendible so that it can be advanced along with ongoing research, by allowing the introduction of new concepts and approaches, such as multi-agent characteristics.

3.1 Architecture

The VITAL system consists of three types of conceptually discrete components: *worlds*, *agents* and *viewers*. These are implemented as separate software applications according to the client-server approach. A world component represents a virtual environment inside which the entire agent system's activity takes place. Agent components represent actors inside an environment. Agents perceive the environment and act upon it according to goal-driven behaviours. Viewers offer the means to human supervisors to observe the environment and all activity inside it in a domain-specific manner. The system's component-based nature is outlined in Figure 1 below:

During system operation, a number of interactions take place between components. In particular, when an agent wants to sense its environment, it requests sensory information from the world component – a *sense request*. The world component then replies, providing the requested information. Similarly, when an agent wants to perform an action, the corresponding agent component provides all necessary action information to the world component – an *action request* – and then the world component responds regarding whether the requested action was successfully applied. In the case of successful action application, the world component sends world change data to all viewer components so that actions are correctly visualised. In addition,

when a viewer needs to build an entirely new visualisation, it requests a full description of the world model – a *world description* request.

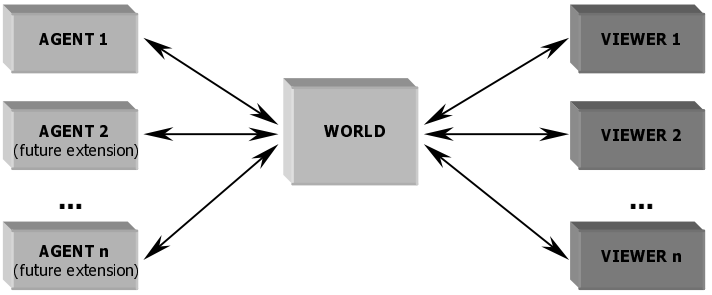


Fig. 1. VITAL system architecture outline

The separation of the world and the agent into two discrete components essentially enables the separation of the logical layer from the physical, real-world layer. This way, agent design can remain focused on the abstract properties and characteristics of the domain addressed and the conceptual specifics of the problem, hence, significantly increasing the system's modelling capacity. On the other hand, the world component can deal with all the realization details, for example, the control of equipment or hardware, giving physical substance to agents' virtual actions.

Adopting the client-server approach directly serves the goal of extendibility, since there is no limitation as to how many agent clients can be connected at any time to a world server. In addition, multiple simultaneous viewer client connections are supported, allowing the system to be observed and monitored from different views, and possibly in different ways, depending on the visualisation capabilities of each viewer client.

3.2 World Modelling

Each component maintains a complete or partial internal representation of the world and, in the case of an agent component, additional properties about itself. The representation used can be either symbolic or object-oriented, according to the component's nature. An additional type of representation, called *pseudo-symbolic*, is used to transfer symbolic facts between applications as well as to translate between symbolic and object-oriented representations.

In the VITAL system, worlds are modelled from the object-oriented point of view as sets of interconnected *locations*. Each location contains one or more *items.* Each item has a *name,* belongs to an item *class* and has *properties*. Agents are also represented as world items. To enrich the modelling scheme with spatial features, each location includes a two- or three-dimensional co-ordinate pair, according to the applications' needs.

This modelling abstraction is adequate to describe a substantial number of different environments: simple mazes, real-world buildings, streets, networks, etc. A world modelled from the object-oriented approach is essentially represented as a partially connected non-directed graph.

To implement this representation as well as to equip it with the necessary management functionality, a specialised class-based hierarchy was defined. According to it, each item has a property of name 'class', which denotes its *item class*, that is, the conceptually wider class of entities the item belongs to. Item classes can be defined according to the application's modelling requirements to any level of abstraction desired. Their selection depends on the properties selected to adequately discriminate types of items in a world. For instance, an agent could be denoted by an item property of name 'class' and value 'agent', whereas a non-agent item could have a 'class' property of value 'object'.

The symbolic modelling methodology defined by the architecture borrows syntactical and semantics elements from predicate logic and logic programming. The VITAL system uses the symbols shown in Table 1 below, with their respective interpretations:

Table 1. World modelling symbols used in the VITAL system

Symbol	**Interpretation**
connects(X1, Y1, X2, Y2)	'(X1, Y1)' and '(X2, Y2)' are connected
at(Item, X, Y)	The location of 'Item' is '(X, Y)'
location(X, Y)	'(X, Y)' is a valid maze location
item(Item)	'Item' is a valid maze item
class(Item, Class)	The class of 'Item' is 'Class'
<property_name>(Item, Value)	'Item' has a property with a name of '<property_name>' and a value of 'Value'

To handle asynchronous network transmission of symbolic information, the architecture introduces a *pseudo-symbolic representation*, according to which, facts and functors are broken down to a series of strings, the first of which being the fact or functor name and the rest arguments. Non-atomic terms, i.e. variables, are represented by an appropriate keyword selected by the agent system designer, for instance, '#VAR'. A *terminating dot* is appended after the last argument to denote that no more arguments should be expected. If a series of facts or functors is to be transmitted, an additional terminating dot is appended after the last fact or functor; thus, two terminating dots should be expected at the end of a series of facts or functors in pseudo-symbolic representation.

3.3 World Server

The world server can be viewed as the central component of the architecture; it provides the grounds on which all action takes place. In addition, it ensures that this action follows specific rules, maintaining consistency throughout the system at all times. Furthermore, from a functional point of view, the world server co-ordinates data interchange between applications ensuring that the agent's model for the world and the viewer's visual representation are valid. The world server is divided into three layers, as shown in Figure 2 below:

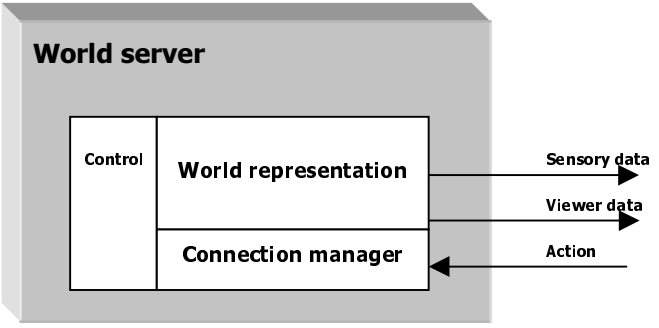


Fig. 2. World server structure

The *world representation* layer contains all data and functionality required to represent and manage the world. The *connection manager* layer is an encapsulation of the server's communications layer, so that the connection management programming interface specifically manages agent and viewer client connections as well as sensory and action requests – a useful programming abstraction offered to agent system developers. Finally, the *control layer* uses the functionality of the world representation layer to receive data from clients, carry out requests and transmit results. The world representation approach used in the world server is object-oriented.

The control layer is responsible for coordinating multiple connections (an agent and one or more viewers). In addition, all data receipt and transmission must be appropriately co-ordinated so that visualisations by viewer clients are consistent with the world model at all times. To achieve that, the control layer operates on a state-dependent basis.

3.4 Agent Client

The agent client is the component with the most vital contribution to the architecture. It stands as an implementation of an actor inside an environment simulated by a world server; it is the ingredient that brings the system to life. It essentially introduces the element of intelligence thanks to innate support for intelligent problem solving. As shown in Figure 3, an agent client consists of an *intelligence layer* and an *interface layer*. The intelligence layer is further divided into the *knowledge base*, the *decision engine* and the set of *sensors* and *effectors*.

Agent clients operate on a *sense-decide-act* cycle. During the *sense stage*, a 'SENSE' keyword is sent to the world server; results received in pseudo-symbolic representation are then processed by sensors and appended to the knowledge base. During the *decision-making stage*, the decision engine reasons upon the contents of the knowledge base and creates a plan for one of the agent's goals. Finally, during the *action* stage, the plan's next action is sent to the world server in pseudo-symbolic representation and effectors apply effects concerning the agent's own internal state.

The knowledge base is the agent's memory. It contains all data that has been perceived by the agent or produced as a result of its reasoning processes. Data in the knowledge base are formulated in a symbolic fashion, thus following the symbolic

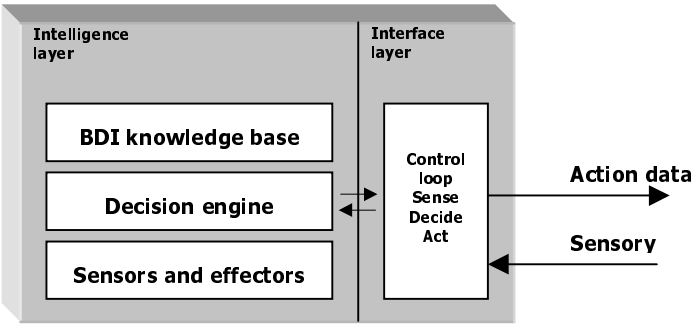


Fig. 3. Agent client structure

modelling approach. Consequently, the knowledge base readily supports the agent'sintelligent reasoning processes. Furthermore, to support intentional reasoning, the knowledge base is structured according to the *BDI (Beliefs-Desires-Intentions)* architecture [4]. An example knowledge base is shown below:

```
[
            beliefs([
              holding([])
            ]),

            desires([
              explore(X, Y),
              pickup(theRedBall),
              drop(theRedBall, 1, 2),
              pickup(theBlueBall),
              drop(theBlueBall, 3, 2),
              move(2, 2)
            ]),

            intentions([

            ])
]
```

Abilities represent the ways an agent can act upon a world. In a VITAL agent, abilities are also defined within the agent's beliefs. In the presented architecture, abilities are defined as (*N, P, E*), where *N* stands for the ability *name*, a functor that identifies it and provides access to all necessary arguments, *P* for the list of *preconditions* and *E* for the list of *effects*. Preconditions are functors that must be included in an agent's knowledge base for an ability to be usable. Effects are functors processed by an agent's effectors to update its beliefs. An example of an ability defined according to the scheme described above follows:

```
ability:        move(X, Y)
preconditions: [at(CX, CY), connects(CX, CY, X, Y)]
```

```
delete:          [at(CX, CY)]
add:             [at(X, Y)]
```

The above example shows the definition for the 'move' ability, that is, the agent's ability to move from one location to another. The preconditions list denotes that the ability is usable only when there is a connection between the agent's current location with co-ordinates (CX, CY) and the new location with coordinates (X, Y). Effects are divided into the *delete* and *add* lists; these denote the facts that must respectively be removed from and added to the agent's beliefs to reflect the new state produced.

The decision engine is responsible for controlling the agent's behaviour; it is an encapsulation of its very intelligence. On every decision-making stage, the engine validates the current plan, and if this fails, or if there is no current plan, the engine attempts to generate a new plan for the goal of top priority. If no plan can be generated for the top goal, the engine attempts to generate a plan for the next one, and keeps doing so until a plan is generated or until no more goals are available.

In VITAL agents, the basic component of the decision engine is the plan generator, or *planner*. The planner consists of two layers: a) a general-purpose means-end planner, and b) a problem-specific second level, which employs heuristics to perform action selection and hence take full advantage of available knowledge into a specific application area, something that significantly reduces computational load and agent response times.

Sensors and effectors encapsulate the agent's sensing and acting functionality; essentially, they are the only access points to its knowledge base. At the end of every sense stage sensors process sensory data received by the world and finally append them to the agent's beliefs. Moreover, after transmission of an action request, effectors update its knowledge base as to reflect its new internal state.

It is important to note that the agent does not have unlimited sensing abilities. What is available to its sensors on any given location is decided upon by the world, following certain rules for vicinity, obstruction, etc. This allows agents to be employed in situations where environment knowledge is obtained gradually. It also gives a certain amount of realism to experimental simulations. The rules are appropriately designed to suit the needs of each application.

Furthermore, an agent's model of the world might not only be a subset of, but inconsistent with the 'real' one, that is, the one maintained by the world server. This may be a desired effect, or the result of erratic operation, most probably due to limited or faulty sensing and reasoning.

3.5 Viewer Client

The viewer client is the encapsulation of the system's visualisation functionality. Similarly to the world server, it maintains an object-oriented world model, and contains a communications layer with sufficient translation functionality. The viewer client's structure is shown in Figure 4 below:

The viewer displays the maze and its contents as a 3D scene, through which a user can navigate freely to obtain a suitable observation angle. Agents are displayed using avatars loaded from VRML files. A number of additional features are also supported,

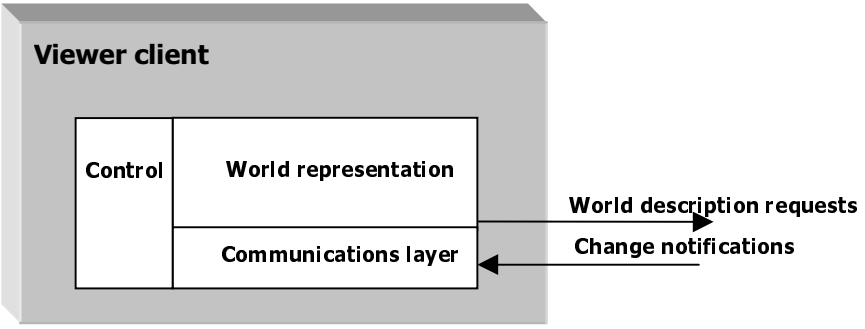


Fig. 4. Viewer client structure

such as lighting, avatar file configuration, as well as locking of the viewpoint on an avatar from various viewing angles, so that agent movement and actions can be easily tracked. The viewer client's user interface as well as the VITAL system in full operation is shown in Figure 5 below.

3.6 Implementation

Reasoning processes within the VITAL agent client have been implemented using SICStus Prolog [15], an implementation of the Prolog language developed at the Swedish Institute of Computer Science (SICS). Apart from being a fully functional Prolog system, offering multiple useful features such as constrained solving, access to operating system resources, parallel solving and many others, SICStus Prolog supports compilation of the Prolog code. Compiled predicates will run faster, using memory more economically. Compiled predicates can be called from within source code in another programming language, such as C++, thanks to an interface provided by the SICStus system. This is an essential feature if reasoning mechanisms built in Prolog are to serve as parts of another Win32 application.

The planner developed for the purposes of the VITAL system is a breadth-first, state-reducing, means-end planner, called *StateExplorer*. Due to its breadth-first nature, it systematically explores a given state space by applying all *available actions* to produce *next* states for a list of *current* ones, hence, the StateExplorer name. The planner was written entirely in the Prolog language; it is defined as a *plan/4* predicate.

The VITAL agent client's second-level planner is implemented as a *plan2/3* Prolog predicate. It is responsible for exploiting domain-dependent knowledge into the nature of the agent's abilities, to minimize computational effort.

The visualisation engine of the VITAL viewer has been implemented using OpenGL. Avatars are loaded from VRML [19] files thanks to a parser implemented using Lex and YACC tools for the Win32 environment, namely the Parser Generator package by Bumble-Bee Software. The parser is capable of processing VRML files containing a subset of the language sufficient to represent scenes and objects exported from major 3D design packages, such as Kinetix 3D Studio and Macromedia Poser.

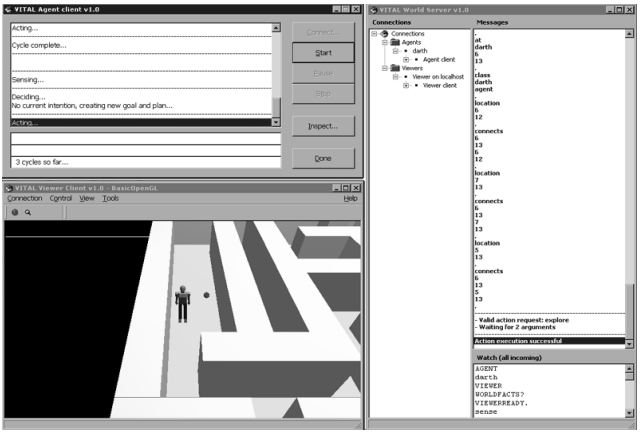


Fig. 5. The VITAL system in full operation

4 Multiple Agent Support and Inter-agent Communication

The VITAL system has recently been extended to support co-existence and simultaneous operation of multiple agents. The new system is named 'mVITAL' (multi-agent VITAL). It enables the development of simulations where intelligent agents communicate through simple speech acts, co-operate and help each other to achieve goals, reason on other agents, exchange beliefs, etc.

As discussed in [1], the agent client is probably the most significant component of the VITAL system, since it introduces the feature of intelligence, thus allowing the properties of agility and dynamic behaviour to emerge. Agent clients in the extended mVITAL system are of even more crucial importance, since they enable the definition of agent societies, introducing the elements of inter-agent interaction, inter-agent communication, co-ordination, and distributed problem solving.

In mVITAL, an agent can identify other agents by sensing their 'class' property and checking if it has a value of 'agent'. In addition, they are able to refer to each other using the value of their 'name' property, just as they refer to themselves using the 'me' keyword. For instance, an agent would believe that 'at(smith, 1, 1)' about agent Smith's position, and 'at(me, 10, 10)' about its own. Social reasoning is from that point on a matter of modelling, and agent interactions emerge as a result of properly defined abilities.

Agents are able to communicate using virtual, non-visual *speech items*. Speech items belong to a special class denoted by a 'speech' value to the item's 'class' property. Speech items have a 'text' property, whose value can be anything an agent wishes to communicate to another, usually text in some natural language. A speech act between two agents can then be modelled as the consecutive application of two actions, one adding a speech item to the world through the effects part, and the other enabling the perception of the item through the preconditions part, thus allowing the receiving agent to respond. Communication can be made even more complex, with

additional speech item properties denoting implied or 'hidden' information, as well as information related to mood, tone, and other expression-related parameters. KQML [6] performatives can be modelled using a suitable set of properties; such a set could include properties such as 'MSG-TYPE', 'VERB', etc, to denote intent, tone, and other information exchanged according to KQML format. To allow more intuitive observation, an extended viewer client can appropriately handle speech-objects and use property values to display communicated text on-screen, reproduce speech using a voice generation engine, etc.

A crucial issue with speech items is that they need to be automatically removed from the world after a certain period of time, to simulate the real world analogy, where a person's speech is heard only while it is spoken. In mVITAL, speech items last a little longer (five seconds) so that all agents in a given range can sense them during a following sense-decide-act cycle. Eventually, five seconds after insertion to the world, speech items are automatically removed by the world server.

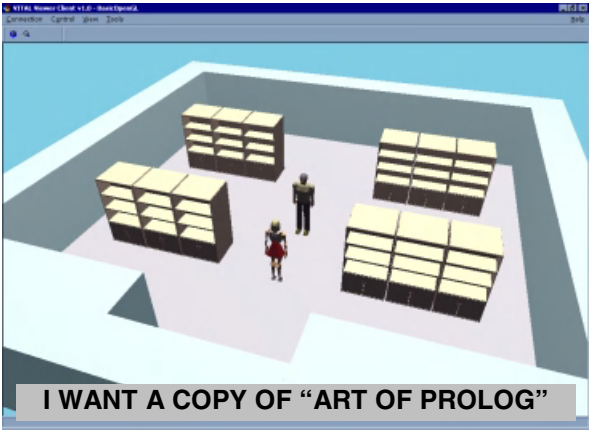


Fig. 6. The client agent requests a copy of "Art of Prolog" from the librarian agent

5 A Working Example

A simple simulation has been modelled in the mVITAL system to demonstrate agent communication and information exchange. The example could originate from the field of computer games or simulation.

The presented scenario involves two agents, a librarian and a library client. The client is looking for a book. In order to get it, it approaches the librarian and asks for it. Then, the librarian sets off and starts looking for the requested book among the library's shelves. If the book is available, the librarian brings it to the client.

According to the VITAL architecture, the above sequence can be modelled as follows: Initially, the client-type agent is committed to requesting a certain book from the librarian. This is denoted by a specific goal in its knowledge base, e.g. 'request_book("some_title")'. Among the effects of this goal, there is the creation of a speech object with a text value of 'I want a copy of "some_title"', or something

similar. After the speech object's creation, the librarian-type agent is able to sense it, and extract book title information from the sensed speech object's text value. Then, a goal is adopted, forcing the librarian to locate the requested book, that is, move close to it. The scenario then follows similarly with the adoption and execution of appropriate goals, forcing the librarian to return with the requested book and pass it to the client, or inform the client that the book is unavailable by creating an appropriate speech object.

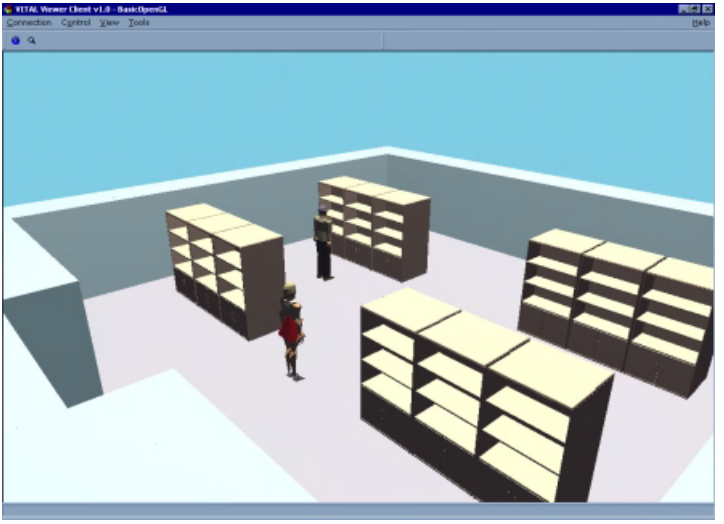


Fig. 7. The librarian is searching for the requested book while the client is waiting

6 Conclusions and Future Work

This paper has been a presentation of the VITAL intelligent agent system, a framework for developing a variety of applications ranging from typical agent-based control systems to VR-based simulations, to IVEs. Despite the fact that numerous issues still need to be addressed for the system to be used in complex evaluation environments such as the RoboCup-Rescue simulator [11], the system is fully functional, and all design requirements have already been fulfilled. In particular, the system is distributed, thus able to exploit the benefits of today's sophisticated networking technologies and the Internet; it employs formal AI techniques - logic programming, planning, intentional reasoning - to support intelligent agent behaviours; it is modular and component-based, enabling the deployment of persistent applications; different types of agents – not necessarily built according to the structure proposed by the architecture, but using the same communication scheme – can be connected to a world server, providing openness and extendibility, as well as enabling dynamic alteration of a simulation's structure and experimentation with other reasoning approaches; the system incorporates sophisticated VR techniques to produce intuitive and believable visualisations; finally, the system comprises a set of

state-of-the-art software applications, built using latest and specialized software engineering technologies.

The VITAL system contributes to original AI and IVE research not by actually extending current methods and approaches, but by providing a robust and well-designed architecture serving as the grounds on which research work and extension can take place. This way, even though VITAL is still at an experimental stage, it eases work towards improvement of the underlying formal approaches, standing as a research testbed where changes are applied in a straightforward manner and results are instantly and intuitively observed; this is the system's most important and original contribution to today's research.

As mentioned in 4, the mVITAL system is still under construction and experimentation. However, evaluation of the VITAL system's performance as a single agent starting point has justified the continuation of the effort towards a multi-agent level, showing that there is strong potential in using MAS technology in the field of IVEs.

Future work directions include, but are not restricted to, making the system more open and configurable, with the ultimate goal being a powerful IVE-authoring tool that will effectively contribute to formalization and standardization in the field. To achieve that, we are already working on the merging of the system with VAL (Virtual Agent Language) [13], to enable abstract and dynamic definitions of agent personalities. VAL is a C/C++-like agent-oriented programming language based on logic programming that was initially developed for the system presented in [14]. Moreover, we are addressing the issue of a world-modelling scheme general enough to enable precise definitions of a variety of scenarios and simulations. Furthermore, we are enriching the agents' planning abilities by introducing explicit temporal and spatial references. In addition, we are investigating more sophisticated visualization and embodiment techniques, as well as the issue of user intervention to simulations through embodiment and presence in the system. Finally, from an implementation point of view, various networking technologies are evaluated in an effort to optimise the system's performance so that real-time execution is guaranteed.

Acknowledgments. This work has been partially supported by the Greek Secretariat of Research and Technology under the PENED 99 project entitled "Executable intensional languages and intelligent multimedia, hypermedia and virtual reality applications", contract no. 99ED265.

References

1. Anastassakis, G.: Intelligent Agents in Vitual Worlds. MSc thesis, UMIST UK (2000)
2. Boulic, R., Capin, T., Huang, Z., Moccozet, L., Molet, T., Kalra, P., Lintermann, B., Magnenat-Thalmann, N., Pandzic, I., Saar, K.,Schmitt, A., Shen, J., Thalmann, D.: The HUMANOID Environment for Interactive Animation of Multiple Deformable Human Characters. Proc. Eurographics '95, Maastricht (1995) 337-348
3. Bratman, M. E., Israel, D. J., Pollack, M. E.: Plans and resource-bounded practical reasoning. Computational Intelligence, 4 (1988) 349-355

4. Bratman, M. E.: Intentions, Plans and Practical Reason. Harvard University Press (1987)
5. Burt, A. D., Fischer, K., Siekmann, J. H.: COGS : Cognitive Architecture for Social Agents. Computational Logic Magazine – The NewsLetter of the European Network in Computational Logic, http://www.cs.ucy.ac.cy/compulog/issue7/areakrr/agents.htm
6. Finin, T., Labrou, Y.: KQML as an Agent Communication Language. Software Agents, Bradshaw, J. M. (ed), MIT Press, Cambridge (1997) 291-316
7. Georgeff, M., Lansky, A.: Reactive reasoning and planning. Proceedings of the Sixth National Conference on Artificial Intelligence (1987) 677-682
8. Grand, S., Cliff, D., Malhotra, A.: Creatures: Artificial Life Autonomous Software Agents for Home Entertainment. Autonomous Agents 97, California USA (1997) 22-29
9. Haddadi, A., Sundermeyer, K.: Belief-Desire-Intention Agent Architectures. Foundations of Distributed Artificial Intelligence, O' Hare, G. M. P., Jennings, N. R., eds. Wiley & Sons, Inc (1996) 169-185
10. Jennings, N. R.: Specification and implementation of a belief desire joint-intention architecture for collaborative problem solving. Journal of Intelligent and Cooperative Information Systems (1993) 289-318
11. Kitano, H.: RoboCup-Rescue: A grand challenge for multiagent systems. Proc. 4th International Conference on MultiAgent Systems (2000)
12. Muller, J. P.: The Design of Autonomous Agents - A Layered Approach. Lecture Notes in Artificial Intelligence, vol. 1177, Springer-Verlag (1996)
13. Panayiotopoulos, T., Anastassakis, G.: Towards a Virtual Reality Intelligent Agent Language. Advances in Informatics (D. Fotiadis, S. Nikolopoulos, eds.), World Scientific (2000) 249-259
14. Panayiotopoulos, T., Katsirelos, G., Vosinakis, S., Kousidou, S.: An Intelligent Agent framework in VRML Worlds. Advances in Intelligent Systems : Concepts, Tools and Applications (S. Tzafestas, ed.), Kluwer Academic Publishers (1999) 33-43
15. SICS Institute: SICStus Prolog User's Manual. SICStus Prolog v3.8 (1999)
16. Sycara, K., Decker, K., Pannu, A., Williamson, M., Zeng, D.: Distributed Intelligent Agents. IEEE Expert, December (1996)
17. Terzopoulos, D., Tu, X., Grzeszczuk, R.: Artificial fishes: Autonomous locomotion, perception, behavior, and learning in a simulated physical world. Artificial Life 1, 4 (1994) 327-351
18. Vosinakis, S., Anastassakis G., Panayiotopoulos, T.: DIVA: Distributed Intelligent Virtual Agents. Extended abstract, presented at the Virtual Agents 99 workshop on Intelligent Virtual Agents, University of Salford UK (1999)
19. VRML Consortium: VRML97 International Standard (ISO/IEC 14772-1:1997). http://www.vrml.org/Specifications/VRML97 (1997)

OilEd: A Reason-able Ontology Editor for the Semantic Web

Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens

Information Management Group, Department of Computer Science,
University of Manchester,
Oxford Road, Manchester M13 9PL, UK
{seanb,horrocks,carole,stevensr}@cs.man.ac.uk,
http://img.cs.man.ac.uk

Abstract. Ontologies will play a pivotal rôle in the "Semantic Web", where they will provide a source of precisely defined terms that can be communicated across people and applications. OilEd, is an ontology editor that has an easy to use frame interface, yet at the same time allows users to exploit the full power of an expressive web ontology language (OIL). OilEd uses reasoning to support ontology design, facilitating the development of ontologies that are both more detailed and more accurate.

1 Introduction

Ontologies have become an increasingly important research topic. This is a result both of their usefulness in a range of application domains [1,2,3], and of the pivotal rôle that they are set to play in the development of the *Semantic Web*

The Semantic Web vision, as articulated by Tim Berners-Lee [4], is of a Web in which resources are accessible not only to humans, but also to automated processes, e.g., automated "agents" roaming the web performing useful tasks such as improved search (in terms of precision) and resource discovery, information brokering and information filtering. The automation of tasks depends on elevating the status of the web from machine-readable to something we might call machine-understandable. The key idea is to have data on the web defined and linked in such a way that its meaning is explicitly interpretable by software processes rather than just being implicitly interpretable by humans.

To realise this vision, it will be necessary to annotate web resources with *metadata* (i.e., data describing their content/functionality). Standardisation proposals for annotation languages have already been submitted to the World Wide Web Consortium (W3C), in particular RDF (Resource Description Framework) and RDF Schema (see [5] for a discussion of the rôles of these languages and of XML/XML Schema). However, such annotations will be of limited value to automated processes unless they share a common understanding as to their meaning. Ontologies, can help to meet this requirement by providing a "representation of a shared conceptualisation of a particular domain" that can be communicated across people and applications [6].

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 396–408, 2001.

RDF Schema (RDFS) itself is already recognisable as an ontology/knowledge representation language: it talks about classes and properties (binary relations), range and domain constraints (on properties), and subclass and subproperty (subsumption) relations. However, RDFS is a relatively primitive language (the above is an almost complete description of its functionality), and more expressive power would clearly be necessary/desirable in order to describe resources in sufficient detail. Moreover, such descriptions should be amenable to *automated reasoning* if they are to be used effectively by automated processes.

These considerations have led to the development of OIL [7], an ontology language that extends RDFS with a much richer set of modelling primitives. A similar RDFS based web ontology language called DAML was been developed as part of the DARPA DAML project [8] and the two languages have now been merged under the name DAML+OIL[1]. OIL has a frame-like syntax, which facilitates tool building, yet can be mapped onto an expressive description logic (DL), which facilitates the provision of reasoning services. OilEd is an ontology editing tool for OIL (and DAML+OIL) that exploits both these features in order to provide a familiar and intuitive style of user interface with the added benefit of reasoning support. Its main novelty lies in the extension of the frame editor paradigm to deal with a very expressive language, and the use of a highly optimised DL reasoning engine to provide sound and complete yet still empirically tractable reasoning services.

Reasoning with terms from deployed ontologies will be important for the Semantic Web, but reasoning support is also extremely valuable at the ontology design phase, where it can be used to detect logically inconsistent classes and to discover implicit subclass relations. This encourages a more descriptive approach to ontology design, with the reasoner being used to infer part of the subsumption lattice (see the case study presented in Section 4); the resulting ontologies contain fewer errors, yet provide more detailed descriptions that can be exploited by automated processes in the Semantic Web. Finally, reasoning is of particular benefit when ontologies are large and/or multiply authored, and also facilitates ontology sharing, merging and integration [9]; considerations that will be particularly important in the distributed web environment.

2 Oil and DAML+OIL

The development of OIL resulted from efforts to combine the best features of frame and DL based knowledge representation systems, while at the same time maximising compatibility with emerging web standards. The intention was to design a language that was intuitive to human users, and yet provided adequate expressive power for realistic applications (many early DLs failed on this second count—see [10]).

The resulting language combines a familiar frame like syntax (derived in part from the OKBC-lite knowledge model [11]), with the power and flexibility

[1] see `http://www.daml.org`

of a DL (i.e., boolean connectives, unlimited nesting of class elements, transitive and inverse slots, general axioms, etc.). The language is defined as an extension of RDFS, thereby making OIL ontologies (partially) accessible to any "RDFS-aware" application.

The frame syntax is less daunting to ontologists/domain experts than a DL style syntax, and it facilitates a modelling style in which ontologies start out simple (in terms of their descriptive content) and are gradually extended, both as the design itself is refined and as users become more familiar with the language's advanced features (see Section 4). The frame paradigm also facilitates the construction and adaption of tools, e.g., the OntoEdit and Protégé editors and the Chimaera integration tool are all being adapted to use OIL/DAML+OIL [12,13, 9].

On the other hand, basing the language on an underlying mapping to a very expressive DL ($\mathcal{SHQ}$) provides a well defined semantics and a clear understanding of its formal properties, in particular that the class subsumption/satisfiability problem is decidable and has worst case ExpTime complexity [14]. The mapping also provides a mechanism for the provision of practical reasoning services by exploiting implemented DL systems, e.g., the FaCT system [15].

OIL extends standard frame languages in a number of directions. One of the key ideas is that an anonymous class description, or even boolean combinations of class descriptions, can occur anywhere that a class name would ordinarily be used, e.g., in slot constraints and in the list of superclasses. For example, in Figure 1 (which uses OIL's "human readable" presentation syntax rather than the more verbose RDFS serialisation), a herbivore is described as an animal that eats only plants or part-of plants. Points to note are that universally quantified (value-type) and existentially quantified (has-value) slot constraints are clearly differentiated, and that the constraint on the eats slot is a disjunction, one of whose components is an anonymous class description (in this case, just a single slot constraint). In addition, it is asserted that the part-of slot is transitive, and that its inverse is the slot has-part. Further details of the language will be given in Section 3, and a complete specification can be found in [7].

```
slot-def part-of
  subslot-of structural-relation
  inverse has-part
  properties transitive
class-def defined herbivore
  subclass-of animal
  slot-constraint eats
    value-type plant OR
      slot-constraint part-of has-value plant
```

Fig. 1. OIL language example

3 OilEd

OilEd is a simple ontology editor that supports the construction of OIL-based ontologies. The basic design has been heavily influenced by similar tools such as Protégé [13] and OntoEdit [12], but OilEd extends these approaches in a number of ways, notably through an extension of expressive power and the use of a reasoner.

However, OilEd is not intended as a replacement for such tools—the current implementation of OilEd is intended primarily as a prototype to test and demonstrate novel ideas, and compromises have been made in the design and implementation. For example, the tool does not provide key functionality for collaborative ontology development such as versioning, integration and merging of ontologies. Similarly, the powerful tailorability and knowledge acquisition aspects of tools such as Protégé have been ignored completely. Rather, the design has concentrated on demonstrating how the frame paradigm can be extended to deal with a more expressive modelling language, and how reasoning can be used to support the design and maintenance of ontologies.

3.1 OilEd Functionality

Basic functionality allows the definition and description of classes, slots, individuals and axioms within an ontology. In general, editing functions are provided through graphical means—mouse driven drop down menus, toolbars and buttons. We will not provide a detailed description of the graphical user interface here, as it is relatively standard (see Figure 2, which provides a screen shot of the editors class definition panel). Instead, we will discuss the novel functionality offered by the tool.

Frame Descriptions. The central component used throughout OilEd is the notion of a *frame description*. This consists of a collection of superclasses along with a list of slot constraints. This is similar to other frame systems. Where OilEd differs, however, is that wherever a class name can appear, a recursively defined, anonymous frame description can be used. In addition, arbitrary boolean combinations of frames or classes (using **and**, **or** and **not**) can also appear. This is in contrast to conventional frame systems, where in general, slot constraints and superclasses must be class names.

As well as being able to assert individuals as slot fillers, several types of constraints on slot fillers can be asserted (these kinds of constraint are sometimes called *facets*). These include *value-type* restrictions (all fillers must be of a particular class), *has-value* restrictions (there must be at least one filler of a particular class), and explicit *cardinality* restrictions (e.g., at most three fillers of a given class). Each constraint has a clearly defined meaning, removing the confusion present in some frame systems, where, for example, it is not always clear whether the semantics of a slot-constraint should be interpreted as a universal or existential quantification.

Class Definitions. A class definition specifies the class name, along with an optional frame description (see above) and a specification of whether the class

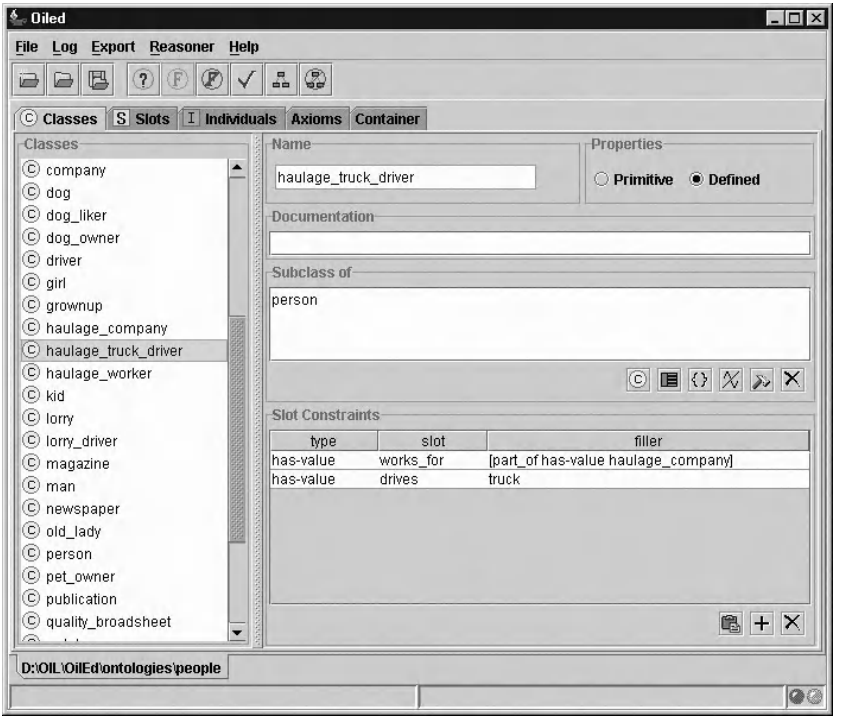


Fig. 2. OilEd Class Panel

is *defined* or *primitive*. If defined, the class is taken to be equivalent to the given description (necessary and sufficient conditions). If primitive, the class is taken to be an explicit subclass of the given description (necessary conditions). In the specification of the OIL language, classes can have multiple definitions. In OilEd, this is disallowed for implementation reasons. Instead classes must have a single definition, but the same effect can be achieved through the use of *equivalence* axioms as discussed below. Ontologies using multiple definitions can be read by the tool. The first definition encountered will be used as the class definition, with any subsequent definitions being translated to the appropriate axioms.

Slot Definitions. A slot definition gives the name of the slot and allows additional properties of the slot to be asserted, e.g., the names of any *superslots* or *inverses*. If r is a superslot of s, then any two objects related via s must also be related via r (i.e., $\mathsf{s}(a,b) \rightarrow \mathsf{r}(a,b)$); if r is an inverse of s, then a is related to b via s iff b is related to a via r (i.e., $\mathsf{s}(a,b) \leftrightarrow \mathsf{r}(b,a)$). Domain and range restrictions on a slot can also be specified. For example, we can constrain the relationship parent to have both domain and range person, asserting that only persons can have, and be, parents. As with class descriptions, the domain and range restrictions can be arbitrary class expressions such as anonymous frames or boolean combinations of classes or frames, again extending the expressivity of traditional

frame editors. Note that in this context, the domain and range restrictions are *global*, and apply to every occurrence of the slot, whether explicit or implicit.

A slot r can also be asserted to be transitive (i.e., $\mathsf{r}(a,b)$ and $\mathsf{r}(b,c) \rightarrow \mathsf{r}(a,c)$), functional (i.e., $\mathsf{r}(a,b)$ and $\mathsf{r}(a,c) \rightarrow b = c$) or symmetric (i.e., $\mathsf{r}(a,b) \rightarrow \mathsf{r}(b,a)$).

All assertions made about slots are used by the reasoner, and may induce hierarchical relationships between classes, e.g., as a result of domain and range restrictions.

Axioms. Another area where the expressive power of OIL/OilEd exceeds that of traditional frame languages/editors is in the kinds of *axiom* that can be used to assert facts about classes and their relationships. As well as standard class definitions (which are really a restricted form of subsumption/equivalence axiom), OilEd axioms can also be used to assert the *disjointness* or *equivalence* of classes (with the expected semantics) along with *coverings*. A covering asserts that every instance of the covered class must also be an instance of at least one of the covering classes. In addition, coverings can be said to be *disjoint*, in which case every instance of the covered class must be an instance of exactly one of the covering classes.

Again, these axioms are not restricted to class names, but can involve arbitrary class expressions (anonymous frames or boolean combinations). This is a very powerful feature, and is one of the main reasons for the high complexity of the underlying decision problem.

Individuals. Limited functionality is provided to support the introduction and description of individuals—the intention within OilEd is that such individuals are for use within class descriptions, rather than supporting the production of large existential knowledge bases (it is supposed that RDF/RDFS will be used directly for this purpose). As an example, we may wish to define the class of Italians as being all those Persons who were born in Italy, where Italy is not a class but an individual.

As the FaCT system does not support reasoning with individuals, they are treated (for reasoning purposes) as disjoint primitive classes. This is not an ideal solution as it does lead to some inferences being lost, in particular those resulting from the interaction between individuals and maximum cardinality constraints. E.g., it would not be possible to infer that Persons who are citizens of Italy, and of no other Country, are citizens of at most one Country. Work is currently underway to extend the FaCT reasoner to deal explicitly with such individuals, so that complete inference can be provided.

Concrete Datatypes. Concrete datatypes (string and integers), along with expressions concerning concrete datatypes (such as min, max or ranges) can also be used within class descriptions. However, the FaCT reasoner does not support reasoning over concrete datatypes, and at present OilEd simply ignores concrete datatype restrictions when reasoning about ontologies. The theory underyling concrete datatypes is, however, well understood [16], and work is also in progress to extend the FaCT reasoner with support for concrete datatypes.

The latest DAML+OIL language release uses the XML Schema type system for the definition of data types. These are not fully supported in our current version of OilEd.

3.2 Reasoning

In addition to the extended expressivity discussed above, OilEd's principal novelty is in its use of reasoning to check class consistency and infer subsumption relationships. Reasoning services are currently provided by the FaCT system, but in principal any reasoner with the appropriate functionality/connectivity could be used.

FaCT is a DL classifier that offers sound and complete reasoning (satisfiability, subsumption and classification) for two DLs: $\mathcal{SHF}$ and $\mathcal{SHQ}$. FaCT's most interesting features are its expressive logic (in particular the $\mathcal{SHQ}$ reasoner), its optimised tableaux implementation (which has now become the standard for DL systems), and its CORBA based client-server architecture [15].

The $\mathcal{SHQ}$ language can completely capture OIL ontologies, with the exception of two recently added features: concrete datatypes (strings, numbers, etc.) and named individuals in class descriptions. As mentioned above, individuals can be dealt with by treating them as pairwise disjoint atomic classes (although with some loss of inferential power), while extending FaCT to deal with OIL's concrete datatypes should be relatively straightforward.

FaCT's optimisations are specifically aimed at improving the system's performance when classifying realistic ontologies. These optimisations lead to performance improvements of several orders of magnitude when compared with older DL and modal logic reasoners, and make the use of reasoning support feasable in spite of the discouraging worst case complexity of the underlying decision problem (ExpTime). The performance improvement is often so great that it is impossible to measure precisely as unoptimised systems are virtually nonterminating with ontologies that FaCT is easily able to deal with [15]. Taking a large medical terminology ontology as an example [17], FaCT is able to check the consistency of all 2,740 classes and determine the complete class hierarchy in about 45 seconds of (700MHz Pentium III) CPU time; unoptimised systems have been run for several weeks without their completing even a single class consistency test.

In the current version of OilEd, reasoning is performed on a "single-shot" basis, i.e., at some suitable point the user connects to the reasoner and requests verification of the ontology. Connection is via FaCT's CORBA based client-server interface, which has the advantage that FaCT servers(s) can be running either locally or remotely, and can provide a service to many OilEd users. Moreover, the FaCT system has reasoning engines for both $\mathcal{SHQ}$ and $\mathcal{SHF}$ knowledge bases, and if both services are available the user can choose to connect to the faster $\mathcal{SHF}$ reasoner to verify an ontology that does not include either inverse slots or cardinality constraints. The current implementation simply informs the user if this is appropriate; future enhancements will include automatic selection of an appropriate reasoning service.

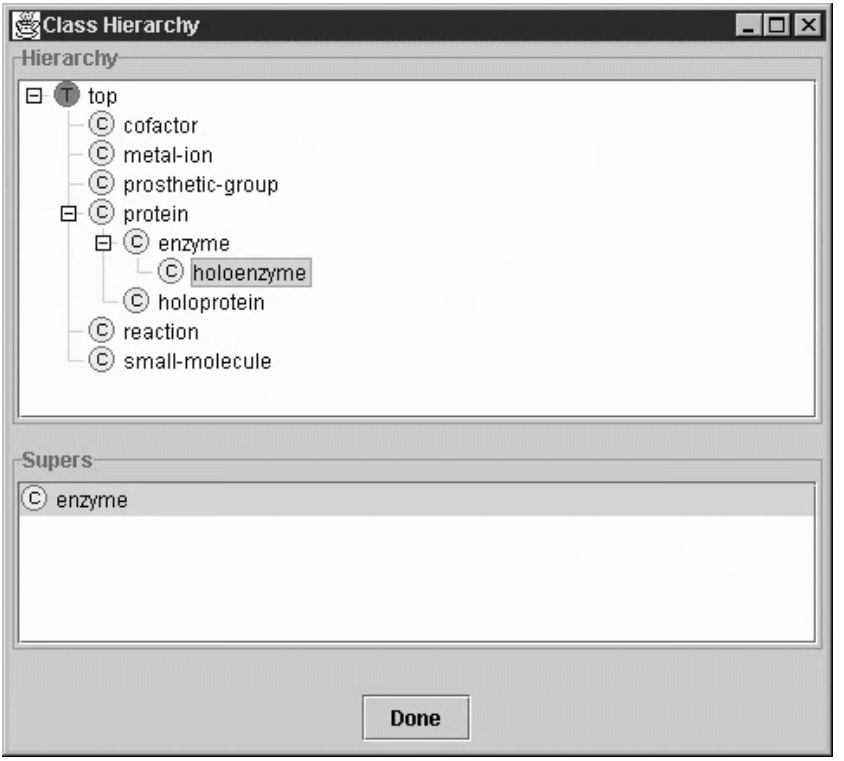


Fig. 3. Hierarchy pre-classification

When verification is requested, the ontology is translated into an equivalent $\mathcal{SHQ}$ (or $\mathcal{SHF}$) knowledge base and sent to the reasoner for classification [18]. OilEd then queries the classified knowledge base, checking for inconsistent classes and implicit subsumption relationships. The results are reported to the user by highlighting inconsistent classes and rearranging the class hierarchy display to reflect any changes discovered. FaCT/OilEd does not provide any explanation of its inferences, although this would clearly be useful in ontology design [19].

Figures 3 and 4 show the effects of classification on (part of) the hierarchy derived from the TAMBIS ontology (see Section 4). When verifying the ontology, a number of new subsumption relationships are discovered (due to the class definitions in the model). In particular we can see that, after verification, holoenzyme is not only an enzyme, but also a holoprotein, and that metal-ion and small-molecule are both subclasses of cofactor.

During subsequent editing, changes to the ontology are not communicated to the reasoner instantaneously, but only when explicitly requested by the user. Future versions of OilEd may incorporate "real-time" reasoning support, but the simple interaction model described here was considered appropriate for the initial prototype.

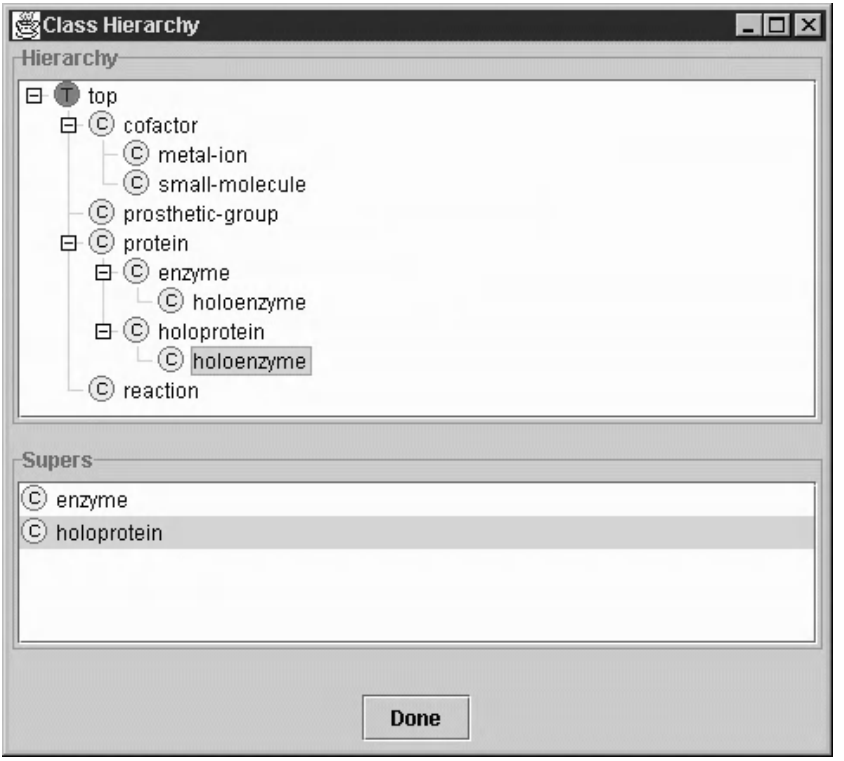


Fig. 4. Hierarchy post-classification

3.3 Export

Although OilEd is primarily intended as an editor for OIL ontologies, the tool will export to a number of formats. These include OIL Standard (the "human-readable" presentation format for OIL that was used in Figure 1), OIL-RDFS (OIL's standard RDFS serialisation) and DAML+OIL (also RDFS). In addition, ontologies can be exported as HTML, facilitating viewing of the ontology without the tool and class hierarchies generated by the classifier can be exported as graphs for viewing with AT&T's Dotty [2] application.

By exporting ontologies as RDFS, it is envisaged that "RDFS-aware" applications will be able to read and interpret OIL ontologies even if they are not fully "OIL-aware". Of course, such applications would be unable to make use of all of the information in the model, but may be able to use, for example, the subclass hierarchies within the ontology. In order to facilitate this, OilEd allows the possibility of explicitly adding all implicit subsumption relationships to the ontology before export, thus making this information available to non-OIL RDFS applications, or even OIL-aware applications that do not employ reasoning.

[2] See `http://www.research.att.com/sw/tools/graphviz/`

4 Case Study: The TAMBIS Ontology

The rôle of ontologies in bioinformatics (the discipline of applying computing to molecular biology) has become prominent in the last few years. Ontologies are used as a mechanism for expressing and sharing community knowledge, to define common vocabularies (e.g., for database annotations), and to support intelligent querying over multiple databases [20]. TAMBIS (Transparent Access to Multiple Bioinformatic Information Sources) is a mediation system that uses an ontology to enable biologists to ask questions over multiple external databases using a common query interface. The ontology is central to the TAMBIS system: it provides a model that queries can be formed against, it drives the query formulation interface, it indexes the middleware wrappers of the component sources, and it supports the query rewriting process [21]. The TAMBIS ontology (TaO) covers the principal concepts of molecular biology and bioinformatics: macromolecules; their motifs, their structure, function, cellular location and the processes in which they act. It is an ontology intended for retrieval purposes rather than hypothesis generation, so it is broad and shallow rather than deep and narrow [20].

The TaO was originally modelled in the DL GRAIL [17]. It was subsequently migrated to OIL in order to (a) exploit OIL's high expressivity so as to maintain a better fidelity with biological knowledge as it is currently perceived; (b) use reasoning support when building and evolving complex ontologies where the knowledge is dynamic and shifting; and (c) be able to deliver the TaO as a conventional frame ontology (with all subsumptions made explicit), thus making it accessible to a wider range of (legacy) applications and collaborators.

The approach to developing the ontology was directly influenced by the range of expressivity that OIL affords, and the capabilities of OilEd itself, particularly its reasoning facilities. The modelling philosophy was to be descriptive, i.e., to model properties and allow as much as possible of the subsumption lattice to be inferred by the reasoner. The design methodology was to first construct a basic framework of primitive foundation classes and slots, working both top down and bottom up, mainly using explicitly stated superclasses. The ontology was then incrementally extended and refined by adding new classes, elaborating slot fillers and constraints, and "upgrading" to defined classes wherever possible, so that class specifications became steadily more detailed and more accurate. This process was guided by subsumption reasoning—when elaborating or changing classes, the reasoner could be used to check consistency and to show the impact on the class hierarchy.

Figure 5 shows a (greatly simplified) fragment of the TaO (using OIL's presentation syntax) that we will use to illustrate this methodology.[3] Originally, holoprotein, enzyme and holoenzyme were all primitive classes, with no slot constraints, and an explicitly asserted class hierarchy: holoprotein and enzyme were subclasses of protein, and holoenzyme was a subclass of enzyme. During the extension and refinement phase, the properties of the various classes were described

[3] The complete ontology can be found at
`http://img.cs.man.ac.uk/stevens/tambis-oil.html`

in more detail: it was asserted that a holoprotein binds a prosthetic-group, that an enzyme catalyses a reaction, and that a holoenzyme binds a prosthetic-group. Several of the classes were also upgraded to being *defined* when their description constituted both necessary and sufficient conditions for class membership, e.g., a protein is a holoprotein if and only if it binds a prosthetic-group. This allows the reasoner to infer additional subclass relationships w.r.t. holoprotein, and in particular that holoenzyme is a subclass of holoprotein. This latter relationship probably would have been missed if the ontology had been hand crafted.

class-def protein

class-def defined holoprotein
  subclass-of protein
  slot-constraint binds *has-value* prosthetic-group

class-def defined enzyme
  subclass-of protein
  slot-constraint catalyses *has-value* reaction

class-def defined holoenzyme
  subclass-of enzyme
  slot-constraint binds *has-value* prosthetic-group

class-def defined cofactor
  subclass-of (metal-ion *or* small-molecule)

disjoint metal-ion small-molecule

Fig. 5. Simplified fragment of TAMBIS ontology

The extension and refinement phase also included the addition of axioms asserting disjointness, equality and covering, further enhancing the accuracy of the model. Referring again to Figure 5, our biologist initially asserted that cofactor was a subclass of both metal-ion and small-molecule (a common confusion over the semantics of 'and' and 'or') rather than being either a metal-ion or a small-molecule. Subsequently, when it was asserted that metal-ion and small-molecule are disjoint, the reasoner inferred that cofactor was logically inconsistent, and the mistake was rectified. Modelling mistakes such as these litter bioontologies crafted by hand.

Other advantages derived from the use of OilEd included:

- The frame-like look and feel of OilEd, and the frame approach of the OIL language, made ontology development much less daunting to our biologist than writing $\mathcal{SHQ}$ logic expressions would have been.
- Clipboard facilities provided by OilEd allowed (parts of) frames to be copied and pasted, making it easy to experiment with new definitions and to maintain a consistent modelling style. E.g., coenzymeA-requiring-oxidoreductase was built by copying nad-requiring-oxidoreductase and changing the constraint on the binds slot from nad to coenzymeA. The reasoner then automatically migrated the class from being a subclass of holoenzyme to being a subclass of coenzyme-requiring-enzyme.

- Class definitions can be as simple as possible yet as complex as necessary. Parts of the TaO are simply primitive frames and slots; other parts are very elaborate and exploit the full expressive power of the OIL language.
- In TAMBIS, the ontology is managed by an ontology server that makes full use of the class definitions, e.g., to classify user generated query classes. However, being able to deliver a static "snapshot" of the ontology in the form of an RDFS taxonomy has proved extremely convenient when working with collaborators who are building ontologies that are in fact simple taxonomies, such as the *Gene Ontology* [22].

5 Conclusion

Ontologies are useful in a range of applications, and will play a pivotal rôle in the Semantic Web, where they will provide a source of precisely defined terms that can be communicated across people and applications. Reasoning with respect to such terms will be important for both the design and deployment of ontologies.

We have presented OilEd, an ontology editor that has an easy to use frame interface, yet at the same time allows users to exploit the full power of an expressive web ontology language (OIL/DAML+OIL). We have also shown how OilEd uses reasoning to support ontology design and maintenance, and presented a case study illustrating how this facility can be used to develop ontologies that describe their domains in more detail and with greater accuracy.

OilEd is a prototype, designed to test and demonstrate novel ideas, and it still lacks many features that would be required of a fully-fledged ontology development environment, e.g., it provides no support for versioning, or for working with multiple ontologies. Moreover, the reasoning support provided by the FaCT system is incomplete for OIL extended with concrete datatypes and individuals, and does not include additional services such as explanation. However, in spite of these shortcomings, OilEd is already sufficiently well developed to be a very useful tool, and to demonstrate the utility of OIL's integration of features from frame, DL and web languages.

References

1. G. van Heijst, A. Schreiber, and B. Wielinga. Using explicit ontologies in KBS development. *Int. J. of Human-Computer Studies*, 46(2/3):183–292, 1997.
2. D. L. McGuinness. Ontological issues for knowledge-enhanced search. In *Proc. of FOIS-98*, 1998.
3. M. Uschold and M. Grüninger. Ontologies: Principles, methods and applications. *K. Eng. Review*, 11(2):93–136, 1996.
4. T. Berners-Lee. *Weaving the Web.* Orion Business Books, 1999.
5. S. Decker *et al.* The semantic web — on the respective roles of XML and RDF. *IEEE Internet Computing*, 2000.
6. T. R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In *Proc. of Int. Workshop on Formal Ontology*, 1993.
7. D. Fensel *et al.* OIL in a nutshell. In *Proc. of EKAW-2000*, LNAI, 2000.

8. J. Hendler and D. L. McGuinness. The DARPA agent markup language. *IEEE Intelligent Systems*, jan 2001.
9. D. L. McGuinness, R. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In *Proc. of KR-00*, 2000.
10. J. Doyle and R. Patil. Two theses of knowledge representation. *Artificial Intelligence*, 48:261–297, 1991.
11. V. K. Chaudhri *et al.* OKBC: A programmatic foundation for knowledge base interoperability. In *Proc. of AAAI-98*, 1998.
12. S. Staab and A. Maedche. Ontology engineering beyond the modeling of concepts and relations. In *Proc. of the ECAI'2000 Workshop on Application of Ontologies and Problem-Solving Methods*, 2000.
13. W. E. Grosso *et al.* Knowledge modeling at the millennium (the design and evolution of protégé-2000). In *Proc. of KAW99*, 1999.
14. I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In *Proc. of LPAR'99*, pages 161–180, 1999.
15. I. Horrocks. Benchmark analysis with fact. In *Proc. TABLEAUX 2000*, pages 62–66, 2000.
16. F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *Proc. of IJCAI-91*, pages 452–457, 1991.
17. A. Rector *et al.* The GRAIL concept modelling language for medical terminology. *Artificial Intelligence in Medicine*, 9:139–171, 1997.
18. S. Decker *et al.* Knowledge representation on the web. In *Proc. of DL 2000*, pages 89–98, 2000.
19. D. McGuinness and A. Borgida. Explaining subsumption in description logics. In *Proc. of IJCAI-95*, pages 816–821, 1995.
20. P.G Baker, C.A Goble, S. Bechhofer, N.W Paton, R. Stevens, and A. Brass. An Ontology for Bioinformatics Applications. *Bioinformatics*, 15(6):510–520, 1999.
21. C. Goble, R. Stevens, G. Ng, S. Bechhofer, N.W. Paton, P.G. Baker, M. Peim, and A. Brass. Transparent Access to Multiple Bioinformatics Information Sources. *IBM Systems Journal*, 40(2), 2001.
22. M. Ashburner *et al.* Gene ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.

Experiments with an Agent-Oriented Reasoning System

Christoph Benzmüller[1], Mateja Jamnik[3], Manfred Kerber[2], and Volker Sorge[1]

[1] Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany
{chris|sorge}@ags.uni-sb.de
http://www.ags.uni-sb.de/
[2] School of Computer Science, The University of Birmingham
Birmingham B15 2TT, England, UK
{M.Jamnik|M.Kerber}@cs.bham.ac.uk
http://www.cs.bham.ac.uk/

Abstract. This paper discusses experiments with an agent oriented approach to automated and interactive reasoning. The approach combines ideas from two subfields of AI (theorem proving/proof planning and multi-agent systems) and makes use of state of the art distribution techniques to decentralise and spread its reasoning agents over the internet. It particularly supports cooperative proofs between reasoning systems which are strong in different application areas, e.g., higher-order and first-order theorem provers and computer algebra systems.

1 Introduction

The last decade has seen a development of various reasoning systems which are specialised in specific problem domains. Theorem proving contests, such as the annual CASC[1] competition, have shown that these systems typically perform well in particular niches but often do poorly in others. First-order provers, for instance, are not even applicable to higher-order problem formulations. Computer algebra systems and deduction systems typically have orthogonal strengths. Whereas many hard-wired integrations of reasoning systems have been shown to be fruitful, rather few architectures have been discussed so far that try to extend the application range of reasoning systems by a flexible integration of a variety of specialist systems.

This paper discusses the implementation of experiments with an agent oriented reasoning approach, which has been presented as a first idea in [BJKS99]. The system combines different reasoning components such as specialised higher-order and first-order theorem provers, model generators, and computer algebra systems. It employs a classical natural deduction calculus in the background to bridge gaps between sub-proofs of the single components as well as to guarantee correctness of constructed proofs. The long term goal is to widen the range of mechanisable mathematics by allowing a flexible cooperation between specialist

[1] CADE ATP System Competitions, see also http://www.cs.jcu.edu.au/~{}tptp/.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 409–424, 2001.

systems. This seems to be best achieved by an agent-based approach for a number of reasons. Firstly, from a software engineering point of view it offers a flexible way to integrate systems. Secondly, and more importantly, the agent-oriented approach enables a flexible proof search. This means that each single system – in form of a pro-active (software) agent – can focus on parts of the problem it is good at, without the need to specify a priori a hierarchy of calls. Currently we still work with a centralised approach and focus on the construction of a single proof object. This means all agents pick up and investigate the central proof object, given in higher-order natural deduction style with additional facilities to abstract from pure calculus layer [CS00]. In case they find that they are applicable in the current proof context they fulfill their task by invoking a tactic by, for instance, calling the external system they encapsulate. After consuming the available resources they come back and make bids in terms of (probably) modified proof objects. Based on heuristic criteria[2] one bid is accepted and executed by the central system while the remaining ones are stored for backtracking purposes. In this sense global cooperation and communication is established in our approach via a central proof object. The benefit is that we have to care only about translations into one single proof representation language, which reduces the proof theoretical and logical issues to be addressed. Furthermore, our central proof object makes use of a human oriented natural deduction format which eases user interaction. For human oriented proof presentation we employ the graphical user interface LOUI [SHB+99] and the proof verbalisation system P.rex [Fie01].

However, extensive communication amongst the agents is currently also a weakness of our system, since too much of the resources are spent on communication. Hence, a future goal is to subsequently reduce this overhead by extending the agents' reasoning capabilities and also by decentralising the approach. A discussion of particular agenthood aspects of our agents will be given in Section 4.

Using the agent paradigm enables us to overcome many limitations of static and hard-wired integrations. Furthermore, the agent based framework helps us to desequentialise and distribute conceptually independent reasoning processes as much as possible. An advantage over hard-wired integrations or even reimplementations of specialised reasoners is that it makes the reuse of existing systems possible (even without the need for a local installation of these systems). Accessing external systems is orchestrated by packages like MATHWEB [FHJ+99] or the logic broker architecture [AZ01]. From the perspective of these infrastructure packages our work can be seen as an attempt to make strong use of their system distribution features.

Our system currently uses about one hundred agents. They are split in several agent societies where each society is associated with one natural deduction rule/tactic of the base calculus. This agent set is extended by further agents encapsulating external reasoners. The encapsulation may be a direct one in case of locally installed external systems, or an indirect one via the MATHWEB framework, which facilitates their distribution over the internet. Employing numerous

[2] For instance, bids with closed (sub)goals are preferred over partial results, and big steps in the search space are preferred over calculus level steps.

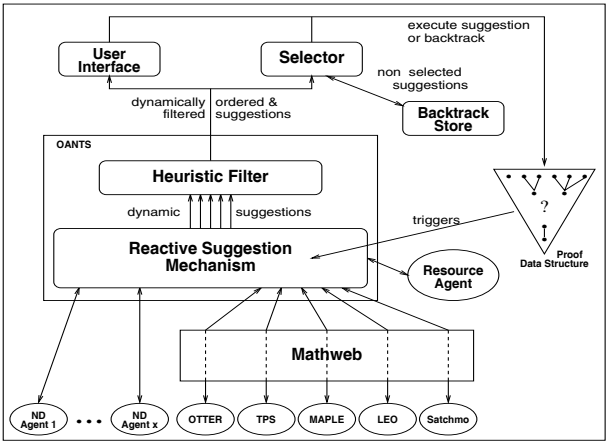


Fig. 1. System architecture.

agents, amongst them powerful theorem provers which are computationally expensive, requires sufficient computation resources. Hence, it is crucial to build the whole system in a customisable and resource adaptive way. The former is achieved by providing a declarative agent specification language and mechanisms supporting the definition, addition, or deletion of reasoning agents (as well as some other proof search critical components and heuristics) even at run-time. For the latter, the agents in our framework can monitor their own performance, can adapt their capabilities, and can communicate to the rest of the system their corresponding resource information. This enables explicit (albeit currently still rudimentary) resource reasoning, facilitated by a specialised resource agent, and provides the basic structures for resource adaptive theorem proving. Further details on the resource and adaptation aspects are addressed in [BS99].

The rest of the paper is structured as follows: Section 2 presents the main components of the system architecture. Experiments with the architecture are sketched in Section 3. In Section 4 we provide an overview of the features of our approach and discuss related work. A conclusion/outlook is given in Section 5.

2 System Architecture

The architecture of our system is depicted in Fig. 1. The core of the system is written in Allegro Common Lisp and employs its multi-processing facilities. The choice of Common Lisp is due the fact that OMEGA, our base system, is implemented in this programming language; conceptually it can be implemented in any multi-processing framework.

Initial problems, partial proofs as well as completed proofs are represented in the **Proof Data Structure** [CS00] and the natural deduction infrastructure provided by the core system, OMEGA [BCF+97].

Our approach builds on the **Reactive Suggestion Mechanism** OANTS [BS01] as a reactive, resource adaptive basis layer of our framework. Triggered

by changes in the proof data structure this mechanism dynamically computes applicable commands with their particular parameter instantiations and calls external reasoners into the current proof state. An important aspect is that all agent computations in this mechanism are de-sequentialised and distributed. The idea of this reactive layer is to receive results of inexpensive computations (e.g., the applicability of natural deduction rules) quickly while external reasoners search for their respective proof steps within the limits of their available resources, until a suggested command is selected and executed. A special resource agent receives performance data from the agents, which monitor their own performance, in order to adjust the system at run time. Heuristic criteria are used to dynamically filter and sort the list of generated suggestions. They are then passed to the selector and/or the user. We give here some sensible heuristic criteria. Does a suggestion close a subgoal? Is a subgoal reduced to an essentially simpler context (e.g., reduction of higher-order problems to first-order or propositional logic)? Does a suggestion represent a big step in the search tree (proof tactics/methods) or a small step (base calculus rules)? Is the suggestion goal directed? How many new subgoals are introduced?

Agents as well as heuristic criteria can be added/deleted/modified at run time. Due to lack of space OANTS cannot be described here in detail; for this we refer the reader to [BS01].

OANTS provides agents that do computations on the basic natural deduction calculus. It also provides agents that invoke additional proof tactics/methods and external reasoning systems. The external reasoning systems are called by the agent-shells indirectly via the MATHWEB system. That is, the agents themselves are realised as concurrent Lisp processes in the core system. These processes activate themselves and make calls to MATHWEB services when their applicability criteria are fulfilled (this contrasts calls by human users to external systems in interactive proof environments).

We extended the approach from [BS01] in the context of our work to integrate partial proofs as results from the external reasoning systems into the overall proof as well as to store different alternative subproofs simultaneously. Moreover, we extended OMEGA's graphical user interface LOUI to be able to display different subproofs of external reasoners as choices for the user.

The **Mathweb system** realises calls to external reasoners which may be distributed over the internet. In our most recent experiments we extensively tested the new ONE-MATHWEB system which is based on a multi-broker architecture. Each broker has knowledge about its directly accessible reasoning systems, and also about urls to other ONE-MATHWEB brokers on the internet. For example, in our experiments the reasoning agents gained access to the computer algebra system MAPLE running in Saarbrücken. For this we simply had to inform the Birmingham MATHWEB broker (which for license reasons cannot offer a MAPLE service locally) about the existence and url of the Saarbrücken broker. The Saarbrücken broker then connects the Birmingham broker (which receives and answers to the requests of the reasoning agents) with the MAPLE service. Currently our system links up with the computer algebra systems MAPLE and GAP running in Saarbrücken, and locally with the higher-order theorem provers LEO and TPS, the first-order theorem prover OTTER (employed also as our

propositional logic specialist), and SATCHMO (employed as a model generator). MATHWEB is described in detail in [FHJ+99].

Once the reactive suggestion mechanism dynamically updates and heuristically sorts the list of suggestions, which are commands together with their particular parameter instantiations, it passes the list on to the **selector**. Its main task is to automatically execute the heuristically preferred command, and hence, initiate an update of the proof data structure. Furthermore, the selector stores the non-optimal, alternative command suggestions in a special store. The information in this store is used when backtracking to a previous state in the proof data structure becomes necessary. Instead of a complete initialisation the reactive suggestion mechanism is then simply initialised with the already computed backtracking information for the current proof context. Backtracking is caused when the reactive layer produces no suggestions or when a user defined maximal depth[3] in the proof data structure is reached.

The **backtrack store** maintains backtracking information for the proof data structure. This information includes representations of the suggestion computations that have been previously computed but not executed. Additionally the store maintains the results of external system calls modulo their translation in the core natural deduction calculus. That is, the immediate translation of external system results is also done by the reactive suggestion layer, and the results of these computations are memorised for backtracking purposes as well. If the system or the user selects to apply the result of an external system, the proof data structure is updated with the translated proof object. Future work will include investigating whether the backtrack store should be merged with the proof data structure. The idea is that each single node in a proof directly maintains its backtracking alternatives instead of using an indirect maintenance via the backtracking store.

The tasks of the **user interface** in our framework are:

1. To visualise the current proof data structure and to ease interactive proof construction. For this purpose we employ OMEGA's graphical user interface LOUI [SHB+99].
2. To dynamically present to the user the set of suggestions, which pop up from the reactive layer to the user, and to provide support for analysing or executing them. This is realised by structured and dynamically updated pop-up windows in LOUI.
3. To provide graphical support for analysing the results of external systems, that is, to display their results after translation/representation in the proof data structure. We achieve this by extending LOUI so that it can switch between the global proof data structure and locally offered results by external systems.
4. To support the user in interacting with the automated mechanism and in customising agent societies at run-time.

From an abstract perspective, our system realises proof construction by going through a cycle which consists of assessing the state of the proof search process,

[3] Iterative deepening proof search wrt. to the maximal depth is conceptually feasible but not realised yet.

evaluating the progress, choosing a promising direction for further search and redistributing the available resources accordingly. If the current search direction becomes increasingly less promising then backtracking to previous points in the search space is possible. Only successful or promising proof attempts are allowed to continue searching for a proof. This process is repeated until a proof is found, or some other terminating condition is reached.

3 Experiments

In this section we report on experiments we conducted with our system to demonstrate the usefulness of a flexible combination of different specialised reasoning systems. Among others we examined different problem classes:

1. Set examples which demonstrate a cooperation between higher-order and first-order theorem provers. For instance, prove:
   $\forall x, y, z.(x = y \cup z) \Leftrightarrow (y \subseteq x \wedge z \subseteq x \wedge \forall v.(y \subseteq v \wedge z \subseteq v) \Rightarrow (x \subseteq v)$
2. Set equations whose validity/invalidity is decided in an interplay of a natural deduction calculus with a propositional logic theorem prover and model generator. For instance, prove or refute:
   a) $\forall x, y, z.(x \cup y) \cap z = (x \cap z) \cup (y \cap z)$
   b) $\forall x, y, z.(x \cup y) \cap z = (x \cup z) \cap (y \cup z)$
3. Concrete examples about sets over naturals where a cooperation with a computer algebra system is required. For instance (gcd and lcm stand for the 'greatest common divisor' and the 'least common multiple'):
   $\{x|x > gcd(10, 8) \wedge x < lcm(10, 8)\} = \{x|x < 40\} \cap \{x|x > 2\}$
   This set is represented by the lambda expression
   $(\lambda x.x > gcd(10, 8) \wedge x < lcm(10, 8)) = (\lambda x.x < 40) \cap (\lambda x.x > 2)$
4. Examples from group theory and algebra for which a goal directed natural deduction proof search is employed in cooperation with higher-order and first-order specialists to prove equivalence and uniqueness statements. These are for instance of the form
   $[\exists \circ .Group(G, \circ)] \Leftrightarrow [\exists \star .Monoid(M, \star) \wedge Inverses(M, \star, Unit(M, \star))]$
   Here $Group$ and $Monoid$ refers to a definition of a group and a monoid, respectively. $Inverses(M, \star, Unit(M, \star))$ is a predicate stating that every element of M has an inverse element with respect to the operation $\star$ and the identity $Unit(M, \star)$. $Unit(M, \star)$ itself is a way to refer to that unique element of M that has the identity property.

We will sketch in the following how the problem classes are tackled in our system in general and how the proofs of the concrete examples work in particular.

3.1 Set Examples

The first type of examples is motivated by the shortcomings of existing higher-order theorem provers in first-order reasoning. For our experiments we used the LEO system [BK98], a higher-order resolution prover, which specialises in extensionality reasoning and is particularly successful in reasoning about sets.

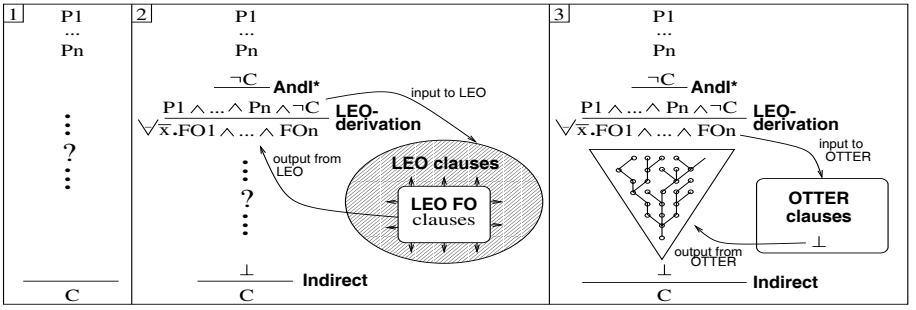


Fig. 2. Agent based cooperation between LEO and OTTER.

Initialised with a set problem LEO tries to apply extensionality reasoning in a goal directed way. On an initial set of higher-order clauses, it often quickly derives a corresponding set of essentially first-order clauses.[4] Depending on the number of generated first-order and other higher-order clauses LEO may get stuck in its reasoning process, although the subset of first-order clauses could be easily refuted by a first-order specialist.

For our examples the cooperation between LEO and the first-order specialist OTTER works as depicted in Fig. 2. The initial problem representation in the proof data structure is described in Part 1 of Fig. 2. The initialisation triggers the agents of the reactive suggestion layer which start their computations in order to produce suggestions for the next proof step.

The agent working for LEO first checks if there is any information from the resource agent that indicates that LEO should stay passive. If not, it checks whether the goal C is suitable for LEO by testing if it is a higher-order problem. In case the problem is higher-order the agent passes the initial problem consisting of the goal C and the assumptions $P_1, \ldots, P_n$ to LEO. While working on the input problem (as indicated by the shaded oval in Part 2 of Fig. 2) LEO derives (among others) various essentially first-order clauses (e.g., $\mathrm{FO}_1 \ldots \mathrm{FO}_n$). For the particular type of cooperation described here, it is important that after a while this subset becomes large enough to be independently refutable. If after consuming all the resources made available by the reactive suggestion layer LEO still fails to deliver a completed proof, it then offers a partial proof consisting of a subset of first-order and essentially first-order clauses (after translation into prenex normal form, e.g., $\forall \overline{x}.\mathrm{FO}'_1 \wedge \ldots \wedge \mathrm{FO}'_n$, where the FO'_i are disjunctions of the literals of FO_i and $\overline{x}$ stands for the sequence of all free variables in the scope). In case LEO's suggestion wins over the suggestions computed by other agents, its partial result is represented in the proof data structure and the reactive suggestion mechanism is immediately triggered again to compute a suggestion for the next possible proof step. Since LEO's partial result is now the new subgoal of the partial proof, first-order agents, like the one working for OTTER, can pick it up and ask OTTER to prove it (see Part 3 of Fig. 2). If OTTER signals a

[4] By essentially first-order we mean a clause set that can be tackled by first-order methods. It may still contain higher-order variables, though.

successful proof attempt before consuming all its given resources, its resolution proof is passed to the natural deduction translation module TRAMP [Mei00], which transforms it into a proper natural deduction proof on an assertion level.[5]

We experimented with 121 simple examples, that is, examples that can be automatically proved by LEO alone. The results showed that the command execution interval chosen by the selector is crucial, since it determines the computation time *ct* made available to the external systems.

- If *ct* is sufficiently high, then the problem is automatically proved by LEO (in case of simple examples that can be solved by LEO alone).
- If *ct* is not sufficient for LEO to come up with a proof, but still enough to produce a refutable subset of essentially first-order clauses, then a cooperative proof is constructed as described above.
- If *ct* is not sufficient to even guarantee a subset of refutable essentially first-order clauses, then the problem is tackled purely on natural deduction level, however not necessarily successfully.

We also solved several examples which cannot be solved with LEO alone. One of them is the concrete example given above, which, to our knowledge, cannot be easily solved by a single automated theorem prover. In our experiments, LEO alone ran out of memory for the above problem formulation, and OTTER alone could not find a proof after running 24 hours in auto mode on a first-order formulation of the problem. Of course, an appropriate reformulation of the problem can make it simple for systems like OTTERto prove this new formulation.

3.2 Set Equations

The second type of set examples illustrates a cooperation between automated natural deduction agents, a propositional prover and a model generator. The proofs follow a well-known set theoretic proof principle: they are constructed first by application of simple natural deduction agents that reduce the set equations by applying set extensionality and definition expansion to a propositional logic statement. This statement is then picked up by an agent working for a propositional logic prover (here we again use OTTER encapsulated in another agent shell with a slightly modified applicability check and a different representation translation approach) and a counter-example agent which employs SATCHMO. The logic statement is then either proved or refuted. Thus, valid and invalid statements are tackled analogously in all but the last step.

In case (2a) of our concrete examples several $\forall_I$ (universal quantification introduction in backward reasoning) applications introduce $(a \cup b) \cap c = (a \cap c) \cup (b \cap c)$ as new open subgoal. Set extensionality gives us $\forall u. u \in (a \cup b) \cap c \Leftrightarrow u \in ((a \cap c) \cup (b \cap c))$. A further $\forall_I$ application and subsequent definition expansions (where $a \cup b := \lambda z.(z \in a) \vee (z \in b)$, $a \cap b := \lambda z.(z \in a) \wedge (z \in b)$, and $u \in a := a(u)$) reduce this goal finally to $(a(d) \vee b(d)) \wedge c(d) = (a(d) \wedge c(d)) \vee (b(d) \wedge c(d))$

[5] While TRAMP already supports the transformation of various machine oriented first-order proof formats, further work will include its extension to higher-order logic, such that also the proof step justified in Fig. 2 with 'LEO-derivation' can be properly expanded into a verifiable natural deduction proof.

which contains no variables and which is a trivial task for any propositional logic prover. In case (2b) we analogously derive $(a(d) \vee b(d)) \wedge c(d) = (a(d) \vee c(d)) \wedge (b(d) \vee c(d))$, but now a model generator agents presents the counter-model $a(d), b(d), \neg c(d)$. That is, it points to the set of all d such that $d \in a$, $d \in b$, but $d \notin c$. Hence, the model generator comes up with a counter-example to the expression in (2b).

We have experimented with an automatically and systematically generated testbed consisting of possible set equations involving $\cap, \cup$, *set-minus* operations up to nesting depth of 5 in maximally 5 variables. We classified 10000 examples with our system discovering 988 correct and 9012 false statements. Naturally, the correct statements are probably also solvable with the cooperation of LEO and OTTER.

3.3 Examples with Computer Algebra

The next type of examples has cross-domain character and requires a combination of domain specific systems. In order to tackle them we added a simplification agent which links the computer algebra system MAPLE to our core system. As an application condition this agent checks whether the current subgoal contains certain simplifiable expressions. If so, then it simplifies the subgoal by sending the simplifiable subterms (e.g., $x > gcd(10, 8)$) via MATHWEB to MAPLE and replaces them with the corresponding simplified terms (e.g., $x > 40$). Hence, the new subgoal suggested by the simplification agent is: $(\lambda x. x > 2 \wedge x < 40) = (\lambda x. x < 40) \cap (\lambda x. x > 2)$. Since no other agent comes up with a better alternative, this suggestion is immediately selected and executed. Subsequently, the LEO agent successfully attacks the new goal after expanding the definition of $\cap$. We have successfully solved 50 problems of the given type and intend to generate a large testbed next.

3.4 Group Theory and Algebra Examples

The group theory and algebra examples we examined are rather easy from a mathematical viewpoint, however, can become non-trivial when painstakingly formalised. An example are proofs in which particular elements of one mathematical structure have to be identified by their properties and transferred to their appropriate counterparts in an enriched structure. The equivalence statement given above in (4) where the unit element of the monoid has to be identified with the appropriate element of the group are in this category. In higher-order this can be done most elegantly using the description operator ι (cf. [And72] for description in higher-order logics) by assigning to the element in the group the unique element in the monoid that has exactly the same properties. In the context of our examples we employed description to encode concepts like the (unique) unit element of a group by a single term that locally embodies the particular properties of the encoded concept itself. If properties of the unit element are required in a proof then the description operator has to be unfolded (by applying a tactic in the system) and a uniqueness subproof has to carried out.

However, an open problem is to avoid unnecessary unfoldings of the description operator as this may overwhelm the proof context with unneeded information.

The idea of the proofs is to divide the problems into smaller chunks that can be solved by automated theorems provers and if necessary to deal with formulae involving description. The ND search procedure implemented in OANTS has the task to successively simplify the given formulae by expanding definitions and applying ND inferences. After each proof step the provers try to solve the introduced subproblems. If they all fail within the given time bound the system proceeds with the alternative ND inferences. The quantifier rules introduce Skolem variables and functions when eliminating quantifications. These are constrained either by the application of a generalised *Weaken* rule, using higher-order unification, or by the successful solution of subproblems by one of the provers, which gives us the necessary instantiation. Problems involving higher-order variables (for which real higher-order instantiations are required) can generally not be solved (in this representation) by first-order provers. However, once an appropriate instantiation for the variables has been computed a first-order prover can be applied to solve the remaining subproblems. Substitutions for introduced Skolem variables are added only as constraints to the proof, which can be backtracked if necessary.

When a point is reached during the proof where neither applicable rules nor solutions from the provers are available, but the description operator still occurs in the considered problem, two theorems are applied to eliminate description. This results in generally very large formulae, which can then again be tackled with the ND rules and the theorem provers.

In our experiments with algebra problems we have successfully solved 20 examples of the described type.

Our experiments show that the cooperation between different kinds of reasoning systems can fruitfully combine their different strengths and even out their respective weaknesses. In particular, we were able to successfully employ LEO's extensionality reasoning with OTTER's strength in refuting large sets of first-order clauses. Likewise, our distributed architecture enables us to exploit the computational strength of MAPLE in our examples remotely over the internet. As particularly demonstrated by the last example class the strengths of external systems can be sensibly combined with domain specific tactics and methods, and natural deduction proof search.

Note that our approach does not only allow the combination of heterogeneous systems to prove a problem, but it also enables the use of systems with opposing goals in the same framework. In our examples the theorem prover and the model generator work in parallel to decide the validity of the current (propositional) goal.

Although many of our examples deal with problems in set theory they already show that the cooperation of differently specialised reasoning systems enhances the strengths of automated reasoning. The results also encourage the application of our system to other areas in mathematics in the future. However, there

is a bottleneck for obtaining large proofs, namely the translation between the different systems involved, in particular, in the presence of large clause sets.

4 Discussion

Our work is related to blackboard and multi-agent systems in general, and to approaches to distributed proof search and agent-oriented theorem proving in particular. Consequently, the list of related work is rather long and we can mention only some of it. We first summarise different facets of our approach which we then use to clarify the differences to other approaches and to motivate our system design objectives. Our system:

(1) aims to provide a cognitively adequate assistant tool to interactively and/or automatically develop mathematical proofs;
(2) supports interaction and automation simultaneously and integrates reactive and deliberative proof search;
(3) maintains a global proof object in an expressive higher-order language in which results of external systems can be represented;
(4) employs tools as LOUI [SHB+99] or P.rex [Fie01] to visualise and verbalise proofs, i.e., communicate them on a human oriented representation layer;
(5) couples heterogeneous external systems with domain specific tactics and methods and natural deduction proof search; i.e., our notion of heterogeneity comprises machine oriented theorem proving as well as tactical theorem proving/proof planning, model generation, and symbolic computation;
(6) reuses existing reasoning systems and distributes them via MATHWEB (In order to add a new system provided by MATHWEB the user has to: a) provide an abstract inference step/command modelling a call to the external reasoner, b) define the parameter agents working for it, and c) (optional) adapt the heuristic criteria employed by the system to rank suggestions. Due to the declarative agent and heuristics specification framework these steps can be performed at run time.);
(7) supports competition (e.g., proof versus countermodel search) as well as cooperation (e.g., exchange of partial results);
(8) follows a skeptical approach and generally assumes that results of external reasoning system are translated in the central proof object (by employing transformation tools such as TRAMP [Mei00]) where they can be proof-checked;
(9) employs resource management techniques for guidance;
(10) supports user adaptation by enabling users to specify/modify their own configurations of reasoning agents at run-time, and to add new domain specific tactics and methods when examining new mathematical problem domains;
(11) stores interesting suboptimal suggestions in a backtracking stack and supports backtracking to previously dismissed search directions;
(12) supports parallelisation of reasoning processes on different layers: term-level parallelisation is achieved by various parameter agents of the commands/ abstract inferences, inference-level parallelisation is supported by the ability

to define new powerful abstract inferences which replace several low level inferences by a single step (a feature inherited from the integrated tactical theorem proving paradigm), and proof-search-level parallelisation is supported by the competing reasoning systems.

Taken individually none of the above ideas is completely new and for each of these aspects there exists related work in the literature. However, it is the combination of the above ideas that makes our project unique and ambitious.

A taxonomy of parallel and distributed (first-order) theorem proving systems is given in [Bon01]. As stated in (12), our approach addresses all three classification criteria introduced there: parallelisation on term, inference, and search level. However, full or-parallelisation is not addressed in our approach yet. This will be future work.

A very related system is the TECHS approach [DF99] which realises a cooperation between a set of heterogeneous first-order theorem provers. Partial results in this approach are exchanged between the different theorem provers in form of clauses, and different referees filter the communication at the sender and receiver side. This system clearly demonstrates that the capabilities of the joint system are bigger than those of the individual systems. TECHS' notion of heterogeneous systems, cf. (5) above, however, is restricted to a first-order context only. Also symbolic computation is not addressed. TECHS [DF99] and its even less heterogeneous predecessors TEAMWORK [DK96] and DISCOUNT [ADF95] are much more machine oriented and less ambitious in the sense of aspects (1)–(4). However, the degree of exchanged information (single clauses) in all these approaches is higher than in our centralised approach. Unlike in the above mentioned systems, our interest in cooperation, however, is in the first place not at clause level, but on subproblem level, where the subproblem structure is maintained by the central natural deduction proof object. Future work includes investigating to what extend our approach can be decentralised, for instance, in the sense of TECHS, while preserving a central global proof object.

In contrast to many other approaches we are interested in a fully skeptical approach, cf. (8) and the results of some external reasoners (e.g., for OTTER TPS, and partially for computer algebra systems) can already be expanded and proof checked by translation in the core natural deduction calculus. However, for some external systems (e.g., LEO) the respective transformation tools still have to be provided. While they are missing, the results of these system, modelled as abstract inferences in natural deduction style, cannot be expanded.

Interaction and automation are addressed by the combination of ILF & TECHS [DD98]. With respect to aspects (6)–(12), especially (10), there are various essential differences in our approach. The design objectives of our system are strongly influenced by the idea to maintain a central proof object which is manipulated by the cooperating and competing reasoning agents, and mirrors the proof progress. This central natural deduction proof object especially eases user interaction on a human oriented layer, cf. (3) and (4), and supports skepticism as described above. In some sense, external systems are modelled as new proof tactics. Extending the background calculus and communication between them is currently only supported via the system of blackboards associated with the

current focus of the central proof object. This relieves us from addressing logical issues in the combination of reasoning systems at the proof search layer. They are subordinated and only come into play when establishing the soundness of contributions of external reasoners by expanding their results on natural deduction layer. A centralised approach has advantages in the sense that it keeps the integration of n heterogeneous systems, with probably different logical contexts, simple and it only requires n different proof (or result) transformation tools to natural deduction arguments. In particular the overall proof construction is controlled purely at the natural deduction layer.

However, experiments indicated that aside from these advantages, the bottleneck of the system currently is the inefficiency in the cooperation of some external systems, especially of homogeneous systems specialised in resolution style proving which cannot directly communicate with each other. Future work therefore includes investigating whether the approach can be further decentralised without giving up much of the simplicity and transparency of the current centralised approach.

With the centralisation idea, we adopted a blackboard architecture and our reasoning agents are knowledge sources of it. In the terminology of [Wei99] our reasoning agents can be classified as reactive, autonomous, pro-active, cooperative and competitive, resource adapted, and distributed entities. They, for instance, still lack fully deliberative planning layers and social abilities such as means of explicit negotiation (e.g., agent societies are defined by the user in OANTS and, as yet, not formed dynamically at run-time [BS01]). In this sense, they are more closely related to the HASP [NFAR82] or POLIGON [Ric89] knowledge sources than to advanced layered agent architectures like INTERRAP [Mül97]. However, in future developments a more decentralised proof search will make it necessary to extend the agenthood aspects in order to enable agents to dynamically form clusters for cooperation and to negotiate about efficient communication languages.

5 Conclusion

In this paper we presented our agent-based reasoning system. Our framework is based on concurrent suggestion agents working for natural deduction rules, tactics, methods, and specialised external reasoning systems. The suggestions by the agents are evaluated after they are translated into a uniform data representation, and the most promising direction is chosen for execution. The alternatives are stored for backtracking. The system supports customisation and resource adapted and adaptive proof search behaviour.

The main motivation is to develop a powerful and extendible system for tackling, for instance, cross domain examples, which require a combination of reasoning techniques with strengths in individual domains. However, our motivation is not to outperform specialised systems in their particular niches. The agent paradigm was chosen to enable a more flexible integration approach, and to overcome some of the limitations of hardwired integrations (for instance, the brittleness of traditional proof planning where external systems are typically

called within the body of proof methods and typically do not cooperate very flexibly).

A cognitive motivation for a flexible integration framework presented in this paper is given from the perspective of mathematics and engineering. Depending on the specific nature of a challenging problem, different specialists may have to cooperate and bring in their expertise to fruitfully tackle a problem. Even a single mathematician possesses a large repertoire of often very specialised reasoning and problem solving techniques. But instead of applying them in a fixed structure, a mathematician uses own experience and intuition to flexibly combine them in an appropriate way.

The experience of the project points to different lines of future research. Firstly, the agent approach offers an interesting framework for combining automated and interactive theorem proving on a user-oriented representation level (and in this sense it differs a lot from the mainly machine-oriented related work). This approach can be further improved by developing a more distributed view of proof construction and a dynamic configuration of cooperating agents. Secondly, in order to concurrently follow different lines of search (or-parallelism), a more sophisticated resource handling should be added to the system. Thirdly, the communication overhead for obtaining large proofs is the main performance bottleneck. More efficient communication facilities between the different systems involved have to be developed. Contrasting the idea of having filters as suggested in [DF99] we also want to investigate whether in our context (expressive higher-order language) abstraction techniques can be employed to compress the exchanged information (humans do not exchange clauses) during the construction of proofs.

Further future work includes improving several technical aspects of the current OMEGA environment and the prototype implementation of our system that have been uncovered during our experiments. We would also like to test the system in a real multi-processor environment, where even the agent-shells for external reasoners can be physically distributed – currently, the agent-shells, which are local, make indirect calls (via MATHWEB) to the external systems. Furthermore, we will integrate additional systems and provide further representation translation packages. The agents' self-monitoring and self-evaluation criteria, and the system's resource adjustment capabilities will be improved in the future. We would also like to employ counter-example agents as indicators for early backtracking. Finally, we need to examine whether our system could benefit from a dynamic agent grouping approach as described in [FW95], or from an integration of proof critics as discussed in [IB95].

Acknowledgements. For advice, support, and encouragement we thank Alan Bundy, Michael Fisher, Malte Hübner, Andreas Franke, and Jürgen Zimmer. This work was supported by EPSRC grants GR/M99644 and GR/M22031 (Birmingham), and the SFB 378 (Saarbrücken).

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A system for distributed equational deduction. In *Proc. of RTA-95*, *LNCS* 914. Springer, 1995.

[And72] P. Andrews. General models, descriptions and choice in type theory. *Journal of Symbolic Logic*, 37(2):385–394, 1972.

[AZ01] A. Armando and D. Zini. Interfacing computer algebra and deduction systems via the logic broker architecture. In [CAL01].

[BCF+97] C. Benzmüller et al. ΩMEGA: Towards a mathematical assistant. In *Proc. of CADE–14*, *LNAI* 1249. Springer, 1997.

[BJKS99] C. Benzmüller, M. Jamnik, M. Kerber, and V. Sorge. Agent Based Mathematical Reasoning. In *CALCULEMUS-99, Systems for Integrated Computation and Deduction*, volume 23(3) of *ENTCS*. Elsevier, 1999.

[BK98] C. Benzmüller and M. Kohlhase. LEO – a higher-order theorem prover. In *Proc. of CADE–15*, *LNAI* 1421. Springer, 1998.

[Bon01] M. Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, in press, 2001.

[BS99] C. Benzmüller and V. Sorge. Critical Agents Supporting Interactive Theorem Proving. *Proc. of EPIA-99*, LNAI 1695, Springer, 1999.

[BS01] C. Benzmüller and V. Sorge. OANTS – An open approach at combining Interactive and Automated Theorem Proving. In [CAL01].

[CAL01] *CALCULEMUS-2000, Systems for Integrated Computation and Deduction*. AK Peters, 2001.

[CS00] L. Cheikhrouhou and V. Sorge. $\mathcal{PDS}$ — A Three-Dimensional Data Structure for Proof Plans. In *Proc. of ACIDCA'2000*, 2000.

[DD98] I. Dahn and J. Denzinger. Cooperating theorem provers. In *Automated Deduction—A Basis for Applications*, volume II. Kluwer, 1998.

[DF99] J. Denzinger and D. Fuchs. Cooperation of Heterogeneous Provers. In *Proc. of IJCAI-99*, 1999.

[DK96] J. Denzinger and M. Kronenburg. Planning for distributed theorem proving: The teamwork approach. In *Proc. of KI-96*, *LNAI* 1137. Springer, 1996.

[FHJ+99] A. Franke, S. Hess, Ch. Jung, M. Kohlhase, and V. Sorge. Agent-Oriented Integration of Distributed Mathematical Services. *Journal of Universal Computer Science*, 5(3):156–187, 1999.

[FI98] M. Fisher and A. Ireland. Multi-agent proof-planning. CADE-15 Workshop "Using AI methods in Deduction", 1998.

[Fie01] A. Fiedler. *P.rex*: An interactive proof explainer. In R. Goré, A. Leitsch, and T. Nipkow (eds), *Automated Reasoning – Proceedings of the First International Joint Conference, IJCAR*, *LNAI* 2083. Springer, 2001.

[Fis97] M. Fisher. An Open Approach to Concurrent Theorem Proving. In *Parallel Processing for Artificial Intelligence*, volume 3. Elsevier, 1997.

[FW95] M. Fisher and M. Wooldridge. A Logical Approach to the Representation of Societies of Agents. In *Artificial Societies*. UCL Press, 1995.

[IB95] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Special Issue of the Journal of Automated Reasoning*, 16(1–2):79–111, 1995.

[Mei00] A. Meier. TRAMP - transformation of machine-found proofs into ND-proofs at the assertion level. In *Proc. of CADE–17*, *LNAI* 1831. Springer, 2000.

[Mül97] J. Müller. A Cooperation Model for Autonomous Agents. In *Proc. of the ECAI'96 Workshop Intelligent Agents III*, *LNAI* 1193. Springer, 1997.

[NFAR82] H. Nii, E. Feigenbaum, J. Anton, and A. Rockmore. Signal-to-symbol transformation: HASP/SIAP case study. *AI Magazine*, 3(2):23–35, 1982.

[Ric89] J. Rice. The ELINT Application on Poligon: The Architecture and Performance of a Concurrent Blackboard System. In *Proc. of IJCAI-89*. Morgan Kaufmann, 1989.

[SHB+99] J. Siekmann et al. LOUI: Lovely OMEGA user interface. *Formal Aspects of Computing*, 11(3):326–342, 1999.

[Wei99] G. Weiss, editor. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, 1999.

[Wol98] A. Wolf. P-SETHEO: Strategy Parallelism in Automated Theorem Proving. In *Proc. of TABLEAUX-98*, *LNAI* 1397. Springer, 1998.

Learning to Execute Navigation Plans*

Thorsten Belker[1] and Michael Beetz[2]

[1] Department of Computer Science
University of Bonn
belker@cs.uni-bonn.de
[2] Department of Computer Science
Munich University of Technology
beetzm@in.tum.de

Abstract. Most state-of-the-art navigation systems for autonomous service robots decompose navigation into global navigation planning and local reactive navigation. While the methods for navigation planning and local navigation are well understood, the *plan execution problem*, the problem of how to generate and parameterize local navigation tasks from a given navigation plan, is largely unsolved. This article describes how a robot can autonomously learn to execute navigation plans. We formalize the problem as a Markov Decision Problem (MDP), discuss how it can be simplified to make its solution feasible, and describe how the robot can acquire the necessary action models. We show, both in simulation and on a RWI B21 mobile robot, that the learned models are able to produce competent navigation behavior.

1 Introduction

Robot navigation is the task of reliably and quickly navigating to specified locations in the robot's operating environment. Most state-of-the-art navigation systems for mobile service robots consist - besides components for map learning and the estimation of the robot's position - of components for global navigation planning and local reactive navigation [KBM98].

Using a map of the operating environment, global navigation planning computes plans for navigating to specified locations in the environment. Navigation plans are typically sequences of discrete actions, mappings from robot states into discrete navigation actions (navigation policies), or paths, sequences of intermediate destinations. Latombe [Lat91] gives a comprehensive overview of these algorithms. Approaches to compute navigation policies using the MDP framework are described in [KCK96,SK95,TBB+98].

Researchers have also investigated a variety of methods for carrying out local reactive navigation tasks. These tasks are those in which the destinations are located in the surroundings of the robot and for which no global (static) map of the environment is used. The reactive navigation methods often employ concurrent, continuous and sensor-driven control processes and a behavior arbitration method that continually combines the output signals of the individual control processes into common control signals for the robot's

* The research reported in this paper is partly funded by the Deutsche Forschungsgemeinschaft (DFG) under contract number BE 2200/3-1.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 425–439, 2001.

drive. Arkin [Ark98] gives a comprehensive introduction to the principles of (behavior-based) reactive control. Other approaches generate trajectories for local navigation tasks based on simple models of the robot dynamics and choose the trajectories with the highest utility [FBT97,Sim96].

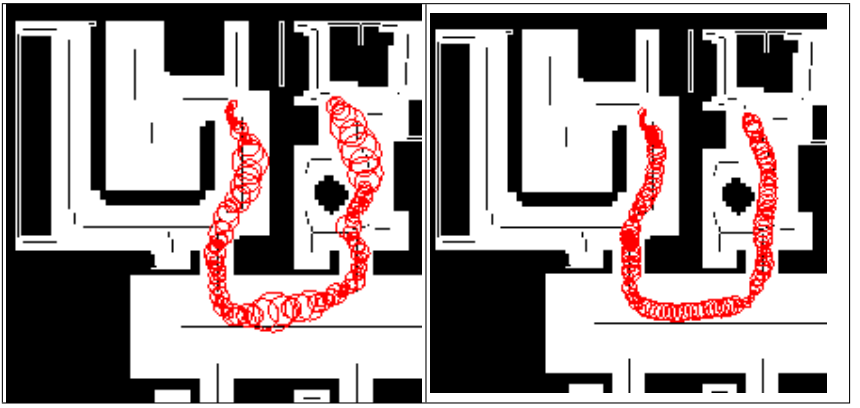


Fig. 1. Two behavior traces produced by the same navigation plan and local navigation module using different plan execution mechanisms. The robot's position is depicted by a circle where the size of the circle is proportional to the robot's translational speed.

While the methods for solving these subproblems are well understood, the *plan execution problem*, the problem of how to combine navigation planning and reactive navigation to produce competent navigation behavior, is largely unsolved. Figure 1 illustrates that the generation of local navigation subtasks and parameterizations of local navigation processes for a given navigation plan has an enormous impact on the performance of the robot. The figure depicts two navigation traces that are generated by the same navigation plan and reactive navigation module using different plan execution schemes. Please refer to [BB00] for a more detailed explanation.

Designing the appropriate plan execution mechanism is very hard and therefore the mechanism is often implemented in an ad-hoc manner and hand-tuned for different applications. Several factors complicate its design. The methods used for navigation planning and local navigation are incompatible. Navigation planning is performed as open-loop control whereas plan execution performs closed-loop control, planning often assumes an abstract discretization of the state and action space whereas reactive navigation deals with continuous processes and asynchronously arriving sensor data streams, planning often works in a drastically reduced state space that ignores dynamic obstacles and the robot's dynamic state, which in turn are handled by reactive navigation. Even worse, the appropriateness of a given plan execution mechanism often depends on the characteristics of the robot and the specifics of the operating environments.

These difficulties suggest that the proper plan execution mechanism should be autonomously learned by the robot. Surprisingly, this computational problem has received little attention so far.

In this paper we describe how a mobile robot can learn to improve its performance by improving the interaction between navigation planning and local navigation. As the main scientific contribution of this article we will show how the navigation plan execution problem can be formulated as a Markov Decision Problem (MDP) in such a way that the solutions to the MDP will produce competent navigation behavior and that the information needed for solving the MDP can be automatically learned by the robot. We will illustrate the power of the approach both in simulation and on a real robot. The implemented plan execution component together with the learning component does not necessarily form a new layer, but can be integrated either in the reactive navigation layer or the path planning layer as done in our implementation.

The remainder of this article is organized as follows: Section 2 states the navigation plan execution problem as a Markov Decision Problem and discusses how this MDP can be simplified to make its solution feasible. In Section 3 we describe how the models necessary to compute a policy for the problem can be autonomously learned. In Section 4 we experimentally demonstrate that the learned action selection policy significantly improves the robot's navigation performance. Section 5 discusses related work.

2 The Plan Execution Problem as an MDP

In this section we briefly introduce the notion of Markov Decision Problems (MDPs) and then formulate the plan execution problem as an MDP.

2.1 MDPs for Stochastic Domains

A Markov Decision Process is a general framework for the specification of simple control problems where an agent acts in a stochastic environment and receives rewards from its environment. A solution of an MDP is a *policy*, a mapping from states into actions that maximizes the expected accumulated reward.

More formally, an MDP is given by a set of states S, a set of actions A, a probabilistic action model $P(S|S,A)$, and a reward function R. $P(s'|s,a)$ denotes the probability that action a taken in state s leads to the state s'. The reward function is a mapping $R : S \times A \rightarrow I\!R$ where $R(s,a)$ denotes the immediate reward gained by taking action a in state s. The property that the effect of an action only depends on the action and the state in which they are executed is called the Markov property. In this formal setting a solution of a Markov Decision Problem is a policy π such that

$$\pi(s) = \underset{a \in A}{\operatorname{argmax}} [R(s,a) + \gamma \int_{s' \in S} P(s'|s,a) V^*(s')] \tag{1}$$

The action $\pi(s)$ in equation 1 is the action that maximizes the sum of the immediate reward $R(s,a)$ for taking action a in state s and $V^*(s')$ the expected future reward after executing the action. $V^*(s')$ denotes the utility of being in state s' when acting according to the optimal policy. The variable γ is the discounting factor which weighs expected future rewards with respect to how far in the future they will occur. Kaelbling [KLC98] gives a detailed discussion of MDPs and POMDPs, a generalization of MDPs where the state is not fully observable.

2.2 Formulating Plan Execution as an MDP

For the following discussion we assume that the robot executes navigation plans only when it is certain about its location in the environment. This is a realistic assumption as we use a localization technique that detects when the robot becomes uncertain about its position in the environment, interrupts navigation tasks in order to relocalize itself, and continues the the navigation tasks after successful relocalization [BFT97] .

Under this assumption we can consider the plan execution problem as an MDP. Unfortunately, the state space of the plan execution problem is infinite and even under reasonable discretizations is the determination of the optimal policy defined by equation 1 computationally too expensive and the policy itself too large to be stored. Therefore, we will heuristically simplify equation 1 and only apply this simplified equation to determine the plan execution policy.

The State Space. For the plan execution problem we characterize states $s \in S$ by the robot's position and orientation, its current sensor readings and the remaining navigation plan given by the path $p = [p_0, p_1, ..., p_n]$ from the robot's current position to its destination. The latter is essential for the plan execution problem, but as paths of arbitrary length in general do not have compact descriptions of fixed length, this also makes the problem difficult.

Ideally the state should also include the dynamic state of the robot, that is its velocity and acceleration. In this paper we still abstract away from the dynamic state and postpone the incorporation of dynamic states into our next research steps. We compensate for this simplification by always letting the robot stop before the next action is chosen.

The Actions. We take the actions that can be executed in a state s to be a point in the remaining navigation plan $p = [p_0, p_1, ..., p_n]$ that can be passed to the reactive navigation component as intermediate target point. This choice of possible actions reflects the following trade-off: While in some circumstances an intermediate target that is farer away from the robot's current position might be advantageous because it allows the reactive navigation system for smoother trajectories, in cluttered environments the target points have to be closer to the robot to be reachable at all. Please note that we do not address the trade-off between short but difficult versus easy but long paths as this is already done by the path planning system. For now we also ignore other actions such as setting the navigation speed, turning in place, and others.

Reward and Probabilistic Action Models. The probabilistic action model $P(S|S, A)$ is a continuous probability distribution in an infinite space. Because this distribution is very complex it is neither possible to specify the distribution by hand nor to learn it e.g. using mixture models. Therefore, the distribution has to be approximated. For this purpose we will assume that an action a can either be successful, in this case the robot will be where it intended to be after executing a, or unsuccessful. In the latter case we assume the robot to be where it started to execute action a. We consider an action a to be unsuccessful when the robot has not reached its target point within t_{to} seconds, that is received a time out. We denote the probability that the robot is timed out when executing

action a in state s as $P(T = true|s, a) = P^+(s, a)$, and the probability that it is not timed out as $P(T = false|s, a) = P^-(s, a)$. While the first assumption (that the robot is where it intended to be after successfully executing an action a) is quite natural, the second assumption (that the robot in case of an unsuccessful action is where it started) is not. However, it reflects the fact that in case of a timeout the robot has wasted time without significant progress.

The immediate reward of executing action a in state s also depends on the success of the action. For a successful execution, the agent receives the reward $R^-(s, a) = -l(a)/v(s, a)$ where $l(a)$ denotes the length of the path to the target point of action a. The term $v(s, a)$ denotes the expected average velocity for the execution of action a in state s. For a timed out action the robot gets an immediate reward of $R^+(s, a) = -t_{\text{to}}$. In both cases the reward is an estimation of the time the robot loses when executing action a. In summary, the robot's expected immediate reward is given by $R(s, a) = \sum_{i\in\{+,-\}} P^i(s, a)R^i(s, a)$.

The Utility Function. The utility of a state s should reflect the time resources that are required to reach the destination from s. The closer s is to the destination, the higher is its utility. Because the state space S we consider is very complex, we cannot determine this utility $V^*(s)$ exactly. We therefore will use the following heuristic approximation of $V^*(s)$:

Let $p = [p_0, p_1, ..., p_n]$ be the path from the robot's current position s to its destination and $l(p)$ be $\sum_{i=1}^{n} |p_{i-1}, p_i|$. Further, let v_{avg} be the robot's average velocity while performing navigation tasks with its best plan execution policy. We can then approximate $V^*(s)$ as $V^*(s) = -l(p)/v_{\text{avg}}$.

Action Selection. Using these simplifications we can replace equation 1 by the following:

$$\pi(s) = \operatorname*{argmax}_{a\in A} \sum_{i\in\{+,-\}} P^i(s, a)(R^i(s, a) + V^i(s, a)) \tag{2}$$

where $P^+(s, a)$, $P^-(s, a)$, $R^+(s, a)$ and $R^-(s, a)$ are as defined above and

$$V^-(s, a) = -\frac{l(p) - l(a)}{v_{\text{avg}}}, V^+(s, a) = -\frac{l(p)}{v_{\text{avg}}} \tag{3}$$

To understand equation 3 recall our assumption that a successful action a leads the robot to the target of a, while an unsuccessful action leaves the robot where it started to execute a. The latter assumption biases the robot to prefer action with a higher probability of success. Please note that we have chosen the discounting factor γ to be 1 as the problem is a finite horizon problem where the agent always reaches an absorbing state after executing a finite number of actions.

In order to get all information needed to solve the plan execution problem as the MDP derived above, the robot has to acquire for each state action pair (s, a) (1) the probability distribution $P(T|s, a)$ and (2) the average velocity the robot will have when executing action a in situation s, $v(s, a)$ if no timeout occurs.

3 Learning the Models

Let us now consider how the function v and the action model $P(T|S, A)$ can be learned. We will apply two alternative approaches: neural network learning and tree induction.

Artificial neural networks are well known as general function approximators. They have therefore often been used to approximate utility and reward functions in reinforcement learning. Tree-based induction methods [BFOS84], on the other hand, have the advantage that they provide in addition to a classification (in case of decision trees) or a value prediction (in case of regression trees) an explanation of the results they produce. The learned tree representations are often valuable resources for human inspection and automated reasoning.

3.1 State Feature Vectors for Plan Execution

Before we can apply any kind of learning algorithm we first have to define the feature language that is used to characterize the concepts and make informed predictions. In order to define an effective feature language we have to identify observable conditions that correlate with the navigation performance.

In our learning experiments we have used the following features: (1) clearance towards the target position, (2) clearance at current position, (3) clearance at target position, (4) minimum clearance on path, (5) curvature of the planned path, (6) average clearance on path, (7) maximal minus minimal clearance on the path, and (8) relative length of path to target position.

The robot's clearance at any position is the distance to the next obstacle in its environment. The clearance towards the target position is the distance to the next obstacle in this direction, but relative to the euclidian distance to the target position. For all k points on the path to the target, we compute the clearance and keep the minimal clearance, the average clearance and the difference between maximal and minimal clearance as features. Another feature is the curvature of the path to the target point a which is $l(a)$ as defined above relative to the euclidian distance to the target point. To compute the relative length of the path towards a, we consider only those a on the path p such that $l(a)$ is smaller than a constant l_{max} (which is in our experiments was 800 centimeters) and the relative length of a as $l(a)/l_{\text{max}}$.

Figure 2 illustrates the features (1) to (4) graphically. The features are relatively simple and can be computed efficiently. We believe that the use of more sophisticated features could probably further improve the robot's performance. The automated learning of more expressive features will be subject of our future research.

3.2 Neural Nets

To learn the action model $P(T|S, A)$ we have used a simple feed forward neural net with sigmoidal activation functions which was trained using an epoch back propagation algorithm. To speed up convergence we have normalized the features described above so that their normalized values are close to 0. Each normalized feature vector was associated with either 0 when no timeout was observed or 1 if a timeout was observed. The output

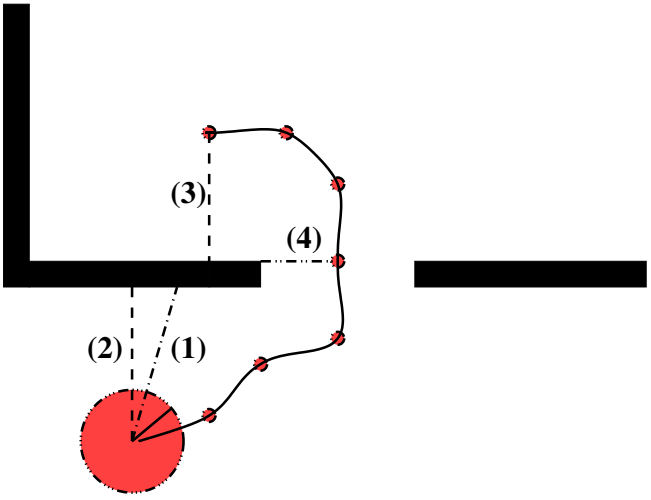


Fig. 2. The features (1) to (4).

of the neural net (after a sufficiently long training) is a value in the interval $[0, 1]$ which can be interpreted as the probability of a timeout.

We have used the same neural network structure to learn the function v. Only the training examples differ. The output values in this case are: $p = v_{\text{cur}}/v_{\text{max}} \in [0, 1]$ where v_{cur} is the robot's current, and v_{max} is the robot's maximal velocity.

3.3 Tree Based Induction

Decision trees [Qui93] can be interpreted as a set of production rules that are well understandable for humans. To learn the action model $P(T|S, A)$ with a decision tree we classify each training example given by the set of features described above as true or as false depending on whether a timeout occured or not. In the decision tree framework a probability can be associated with each classification like this: if n is the number of training examples that are mapped to a decision tree leaf l that is associated with the classification c and m is the number of examples in this set that are classified as c, we can associate the probability $p = m/n$ with this classification (given the observed features).

To learn the function v is a regression and not a classification task. Regression trees [BFOS84] are tree based function approximators and can be applied to regression tasks. Regression trees are similar to decision trees, but differ from them in that leaf nodes are not associated with classifications, but with real values. A regression tree like a decision tree can be translated in a set of production rules.

To build a regression tree a set of training examples associated with a node in the tree is split to minimize some given impurity measure. This impurity measure often is the empirical variance of the output value of the function to learn. For example, to split a set of n examples, S, with a variance in the output value σ^2, into two sets S_1 and S_2, a split is chosen that minimizes $n_1\sigma_1^2 + n_2\sigma_2^2$ where n_i is the number of examples in

S_i and σ_i^2 is the variance in the set S_i with respect to the output value. This process is iterated recursively until some given stopping criterion is met.

We have used the following stopping criterion: For each split we have tested if the split reduces the variance significantly. If the best split does not reduce the variance significantly we stop growing the tree at that node. Whether a split reduces the variance significantly can be tested using a bootstrapping t-test [Coh95]. The only parameter of this test is a lower bound θ of the estimated probability p, we will call *significance level*, that the split really reduces the variance. If $p > \theta$ we say that the split reduces the variance significantly. The choice of θ allows us to trade-off the prediction accuracy of the tree (on the training examples) against its complexity.

Regression trees are limited in that they split the input space into regions with boundaries that are parallel to the main axis of the input space. This limitation can be overcome by using multivariate splits. For our experiments however we have used a simple implementation that was restricted to main axis parallel splits.

4 Experimental Results

In this section we will demonstrate (1) that the policy defined by equation 2 can be used to execute a global navigation plan quickly and reliably and (2) that the necessary models can be learned autonomously. We have performed both, simulator experiments and experiments on a real robot, to show that the learned action selection improves the robot's performance substantially and significantly.

The experiments are run on an RWI B21 mobile robot platform and its simulator. For navigation we have used a re-implementation of the path planning module described in [TBB$^+$98], which considers path planning as an MDP. As reactive navigation component we have used the one described in [FBT97]. It generates possible trajectories for local navigation tasks based on simple models of the robot dynamics and chooses the trajectories with the highest utility.

The setup of the experiments is as follows. After the learning phase, in which the robot has acquired the action models needed, a default and the learned plan execution method are compared. The default method chooses the next target point randomly between 1 and 8 meters ahead on the planned path. Both methods are compared based on a sequence of k navigation tasks that is executed n times. We then test whether the learned action selection mechanism improves the robot's performance significantly. This is done with a bootstrapping t-test [Coh95].

4.1 Learning the Models

To learn the velocity function v and the timeout probability $P(T|S, A)$ we have generated some training and test data using the random action selection as described above and the simulator. We used a set of 5279 training- and 3266 test examples for the classification task (was the robot timed out when performing a given action?) and 4203 training- and 2531 test examples for the regression task (what was the robot's average velocity when performing an action?). This data volume corresponds to collecting data from robot's runs that take about 24 hours. For the regression task we only considered examples where the robot was not timed out.

In the experiments we will present here, we have not used new experience to incrementally improve the models mainly to simplify the evaluation of the experiments.

Learning the Models with Neural Nets. For the classification task as well as for the regression task we trained a neural network with 8 nodes in the input layer, 8 nodes in the hidden layer and 1 node in the output layer. We performed epoch back propagation to train both neural nets.

For the classification task we used a learning rate of 0.8 and a momentum term of 0.9. After 203590 iterations (a night) we had 88.14% of the training- and 88.77% of the test examples correctly classified.

For the regression task we used a learning rate of 0.8 and a momentum term of 0.95. After 406360 iterations (about 24 hours) we got an absolute error of 3.698 cm/s on the training- and of 3.721 cm/s on the test set where 60 cm/s is the maximal velocity of the robot.

Learning the Models with Tree Induction. We used the same data to learn a decision and a regression tree. Table 1 gives the statistics of the training error, the test error and the number of generated rules depending on the significance level used in the stopping criterion for the decision tree learning.

Table 1. The training error, the test error and the number of generated rules depending on the significance level used in the stopping criterion for the decision tree learning.

LEVEL	TRAIN. ERROR	TEST ERROR	RULES
0.5	6.6%	13.2%	301
0.6	7.4%	12.6%	202
0.7	8.0%	12.7%	146
0.8	9.6%	12.9%	72
0.9	12.0%	14.0%	21
0.95	12.3%	14.1%	12

Table 2 gives the same statistics for the regression tree where the training and test errors are absolute errors in cm/s.

In our experiments we have chosen the rules generated from the trees grown with the 0.95 significance level. These rules are well understandable when inspected by a human operator. The trees are grown within about two minutes.

Table 2. The absolute training and test error and the number of generated rules depending on the significance level used in the stopping criterion for the regression tree learning.

LEVEL	TRAIN. ERROR	TEST ERROR	RULES
0.5	2.71 cm/s	3.76 cm/s	743
0.6	3.33 cm/s	3.76 cm/s	256
0.7	3.68 cm/s	3.88 cm/s	96
0.8	3.98 cm/s	4.11 cm/s	48
0.9	4.27 cm/s	4.30 cm/s	29
0.95	4.33 cm/s	4.37 cm/s	25

4.2 Simulator Experiments

In the simulator experiments we compared the learned and the random action selection mechanism based on a sequence of 14 navigation tasks in a university office environment that was executed 18 times. Figure 3 shows the set of tasks that had to be performed.

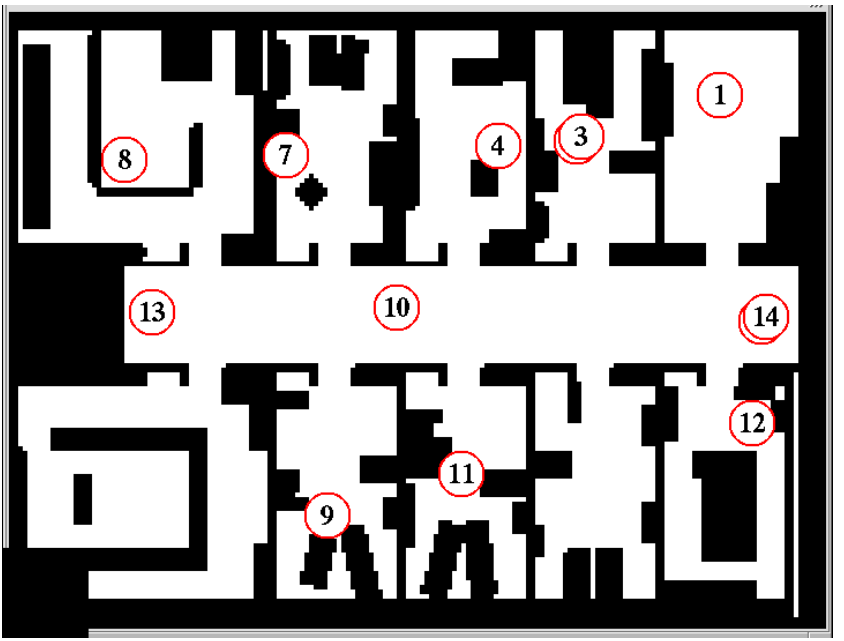


Fig. 3. The set of target points that define the 14 navigation tasks performed in each iteration of the experiment (three of the points define two navigation tasks depending on the starting position of the robot).

Experiment 1. In the first experiment we compared the performance of the two action selection mechanisms, where the models are learned using neural networks. The average

time needed to complete all tasks is 1380 seconds with random action selection and 1107 seconds with the learned action selection. This is a performance gain of 19.8%. The probability of a reduced average duration (computed with a bootstrapping t-test [Coh95]) is 0.998, and the performance gain therefore clearly significant. The probability that the standard deviation in the robot's behavior can be reduced when using the learned action selection mechanism is 0.98. Table 3 summaries these results.

Table 3. The results of Experiment 1.

average time with random action selection	1380.44 s
average time with learned action selection	1107.78 s
performance gain	19.75 %
probability of reduced average duration	99.98 %
probability of reduced variance	97.53 %

Experiment 2. In the second experiment we compared the learned action selection using the tree structured models with the random action selection. With the learned action selection the robot on average needed 925 seconds to complete the whole sequence, in contrast to 1380 seconds with the random action selection. That is a reduction of 33.0 %. The probability that the average time needed to execute the sequence of navigation tasks was reduced is 1. Furthermore, the probability that the standard deviation of the execution times is reduced is 0.99. The results are summarized in Table 4.

Table 4. The results of Experiment 2.

average time with random action selection	1380.44 s
average time with learned action selection	925.39 s
performance gain	32.96 %
probability of reduced average duration	100.00 %
probability of reduced variance	99.03 %

Experiment 3. A purely random action selection seems to be a quite weak standard for the performance comparisons. However, as we will show in this experiment, this is not the case. To demonstrate this, we compared a deterministic action selection with the random one: the robot always selects as next target point the last point on the path that is still visible from its current position. This is more or less the strategy that is used in our regular navigation system. Surprisingly, the performance is about 11.6 % worse with the deterministic strategy compared to the random action selection. The robot in average needs 1541 seconds for the 18 navigation tasks compared to the 1380 seconds with the random action selection. Therefore the random action selection is significantly better than the deterministic one (with respect to a significance level of 99 %). However, the standard deviation is significantly smaller, 115.2 seconds compared to 237.9 seconds, with respect to the same significance level.

4.3 Experiments with the Mobile Robot

In the experiments performed on the mobile robot, we test how well the models learned in simulation generalism when tasks different from those considered in the training phase have to be performed by a real robot. This is an interesting question because it is unrealistic to require state-of-the-art research platforms to perform the time-consuming learning without keeping them under surveillance. Therefore it should be possible to learn at least part of the models in simulation.

The robot was to execute a sequence of 5 navigation tasks 18 times both with random action selection and with the informed action selection described above. The experiments were carried out in a populated university office environment. Figure 4 shows the environment and the tasks that have to be performed within that environment.

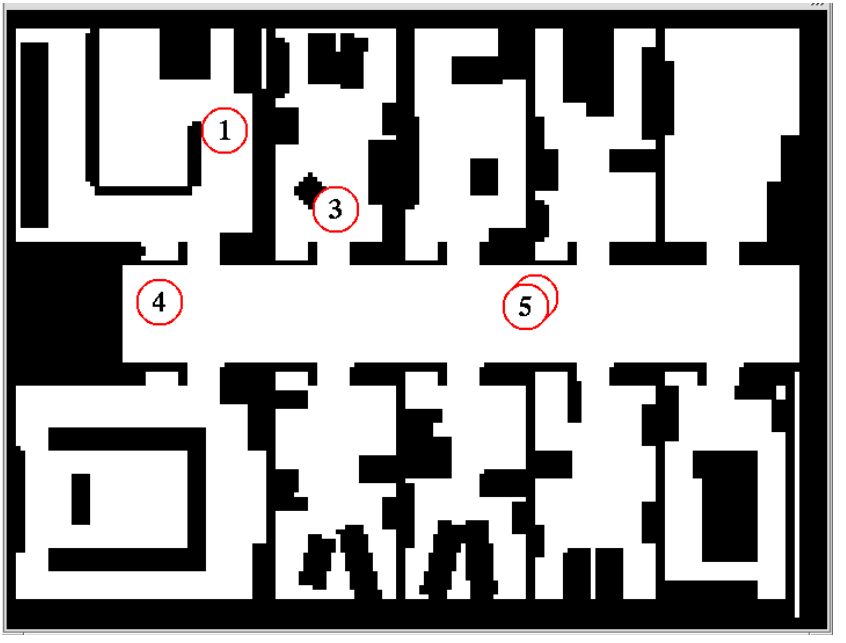


Fig. 4. The set of target points that define the 5 navigation tasks performed in each iteration of the experiment with the real robot.

Experiment 4. When using neural nets to learn the models, the robot needed 362 seconds in average to perform the five tasks, opposed to 420 seconds with the random action selection. This is a performance gain of 13.8%. The probability that the average time is reduced when using the action selection mechanism of equation 2 is 0.999. The probability that the standard deviation was reduced is 0.94. The robot's performance can be summarized by descriptive statistics shown in Table 5.

Experiment 5. In the last experiment we have compared the performance of the robot using equation 2 and decision/regression trees for the models with its performance

Table 5. The results of Experiment 4.

average time with random action selection	420.06 s
average time with learned action selection	362.00 s
performance gain	13.82 %
probability of reduced average duration	99.98 %
probability of reduced variance	94.59 %

using random action selection: The average time needed to execute the sequence of five navigation tasks is 306 seconds with the learned models as opposed to 420 seconds with the random action selection. This is a reduction of 27.1%. The probability that the average time has been reduced with the learned action model is 1 and and the probability that the standard deviation was reduced is 0.99. The descriptive statistics of this experiment are summarized in Table 6.

Table 6. The results of Experiment 5.

average time with random action selection	420.06 s
average time with learned action selection	306.17 s
performance gain	27.11 %
probability of reduced average duration	100.00 %
probability of reduced variance	99.03 %

4.4 Discussion of the Experimental Results

The experiments show that the policy defined by equation 2 can be used to solve the plan execution problem and that the necessary models can be autonomously learned by the robot. In the domain, tree structured models perform slightly better than neural nets. However, these results do not support a general comparison of the two learning methods as we have used fairly simple implementation of both methods.

A main problem for both learning methods is the noisy data. It is mainly caused by two facts: (1) The local navigation component has a lot of special purpose behaviors to react on exceptional situations, but the learning component currently can neither control nor even monitor their activation. This makes it rather difficult to predict the behavior of the reactive component and causes a lot of variance and noise in the training data. (2) The feature language is not expressive enough to fully discriminate situations where the robot is successful from those where it is not (partly because of (1)). We will address both problems in our future work.

5 Related Work

In this paper we have applied the MDP framework to the problem of navigation plan execution. To the best of our knowledge, this is a new application of the MDP framework. However, it has been applied to mobile robot navigation before, for example to plan navigation policies [KCK96,SK95]. Reinforcement learning and especially Q-learning

have been applied to learning reactive navigation [Thr96,RS97]. Often reinforcement learning is combined with neural network learning as in the case of [Thr96], more seldom with the use of instance-based methods as in the case of [RS97].

More recently reinforcement learning has been combined with regression trees. Sridharan et al. [ST00] applied Q-learning with regression trees in multi-agent scenarios. They point out that regression trees have been superior to neural networks both in the policies learned and the reduced training effort. Wang et al. [WD99] discuss the use of regression trees in combination with TD(λ)-learning to solve job-shop-scheduling problems. They utter the hope that regression tree learning might be suitable to become the "basis of a function approximation methodology that is fast, gives good performance, requires little-or-no tuning of parameters, and works well when the number of features is very large and when the features may be irrelevant or correlated".

Balac et al. [BGF00] report the use of regression trees for (again purely reactive) robot control, but they do not compare regression trees to other function approximators. The use of decision trees - more specifically the use of confidence factors in decision tree leafs - for robot control is reported in [SV98].

Our work is also related to the problem of learning actions, action models or macro operators for planning systems. Schmill *et al.* recently proposed a technique for learning the pre- and postconditions of robot actions. Sutton *et al.* [SPS99] have proposed an approach for learning macro operators in the decision-theoretic framework using reinforcement learning techniques.

6 Conclusion

We have discussed how the problem of navigation plan execution can be formalized in the MDP framework and how the formalization can be heuristically simplified to make its solution feasible. This formalization allows for the application of standard learning techniques like neural networks and decision/regression trees. The approach has been tested both in simulation and on a RWI B21 mobile robot to demonstrate that it improves the robot's behavior significantly compared to a plan execution method that selects actions randomly from a set of admissible actions.

These results are important for the application of autonomous mobile robots to service tasks. Using the technique described in this paper we can build navigation systems that employ state-of-the-art global navigation planning and local execution techniques and let the robots themselves learn the heuristic, environment- and robot-specific interface between the components. Even with a very restricted feature language and a much reduced action set we could achieve significant performance gains. The next steps on our research agenda are the development of a more expressive and adequate feature language, the introduction of more powerful actions such as turning in place, varying the speed, etc, and the development of more accurate approximations of the plan execution problem for which the necessary models can be learned in moderate time (for regression/decision tree within a few minutes) and the necessary training data can be collected in a few days (in our case within 24h in simulation).

References

[Ark98] R.C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, 1998.

[BB00] M. Beetz and T. Belker. Environment and task adaptation for robotic agents. In *Procs. of the 14th European Conference on Artificial Intelligence*, 2000.

[BFOS84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth, Inc., Belmont, CA, 1984.

[BFT97] W. Burgard, D. Fox, and S. Thrun. Active mobile robot localization. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[BGF00] M. Balac, D.M. Gaines, and D. Fisher. Using regression trees to learn action models. In *Proceedings 2000 IEEE Systems, Man and Cybernetics Conference*, 2000.

[Coh95] P. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA, 1995.

[FBT97] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 4(1), 1997.

[KBM98] D. Kortenkamp, R.P. Bonasso, and R. Murphy, editors. *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, Cambridge, MA, 1998.

[KCK96] L. Kaelbling, A. Cassandra, and J. Kurien. Acting under uncertainty: Discrete Bayesian models for mobile-robot navigation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1996.

[KLC98] L. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.

[Qui93] R.J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, San Mateo, California, 1993.

[RS97] A. Ram and J. Santamaria. Continous case-based reasoning. *Artificial Intelligence*, 90(1-2):25–77, 1997.

[Sim96] Reid Simmons. The curvature-velocity method for local obstacle avoidance. In *International Conference on Robotics and Automation*, 1996.

[SK95] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1995.

[SPS99] R. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[ST00] M. Sridharan and G. J. Tesauro. Multi-agent q-learning and regression trees for automated pricing decisions. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[SV98] P. Stone and M. Veloso. Using decision tree confidence factors for multiagent control. In *RoboCup-97: The First Robot World Cup Soccer Games and Conferences*. 1998.

[TBB+98] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schmidt. Map learning and high-speed navigation in RHINO. In D. Kortenkamp, R.P. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, Cambridge, MA, 1998.

[Thr96] S. Thrun. An approach to learning mobile robot navigation. *Robotics and Autonomous Systems*, 15:301–319, 1996.

[WD99] X. Wang and T. Dietterich. Efficient value function approximation using regression trees. In *Proceedings of the IJCAI-99 Workshop on Statistical Machine Learning for Large-Scale Optimization*, 1999.

DiKe – A Model-Based Diagnosis Kernel and Its Application*

Gerhard Fleischanderl[1], Thomas Havelka[3], Herwig Schreiner[1], Markus Stumptner[2], and Franz Wotawa[3]**

[1] Siemens Austria, Program and System Engineering, Erdbergerlände 26, A-1030 Vienna, Austria, {Gerhard.Fleischanderl,Herwig.Schreiner}@siemens.at
[2] University of South Australia, School of Computer and Information Science, Mawson Lakes Boulevard 1, 5092 Mawson Lakes SA, Australia, mst@cs.unisa.edu.au
[3] Technische Universität Wien, Institut für Informationssysteme, Favoritenstraße 9–11, A-1040 Vienna, Austria,{havelka,wotawa}@dbai.tuwien.ac.at

Abstract. This paper describes the DiKe model-based diagnosis framework, which incorporates multiple diagnosis engines, multiple user-level system description languages, a theorem prover, and a graphical user interface to provide an integrated toolset for the development of model-based diagnosis applications. The framework has been used for representing a number of application domains. We present the AD2L language, the main user language for the system geared towards use by non-specialists, and discuss use of DiKe in various domains.

1 Introduction

Model-based Diagnosis and Model-Based Reasoning are two areas of knowledge-based systems research that grew out of the late 1980s' disenchantment with traditional rule-based expert system technology. The goal was to avoid the brittleness of the latter systems by using a Reasoning from First Principles approach, and the maintenance issues by providing high-level representation languages with unambiguous formal semantics. Overall that goal can be considered to have been attained as model-based systems are being employed in a variety of application areas. On the other hand, whereas rule-based tools are still widespread and used by many practitioners on actual applications projects, the model-based approach has so far not really moved out of the academic world. What applications there are are quite successful but still require the attendance of a research team to develop and implement system descriptions and implement or at least tune special-purpose reasoning engines. There is no widespread understanding of the principles, acceptance of the advantages, or support from a user community as with the continuing "grassroots" existence of various development environments for rule-based systems.

Our goal is to facilitate the development of model-based diagnosis into a technology that can be readily used even by individuals without a formal training in AI techniques.

* This work was partially supported by the Austrian Science Fund project N Z29-INF, Siemens Austria research grant DDV GR 21/96106/4, and the Hochschuljubiläumsstiftung der Stadt Wien grant H-00031/97.

** Authors are listed in alphabetical order.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 440–454, 2001.

To this aim we have built an integrated diagnosis toolkit that provides different types of diagnosis engines, graphical user interfaces, and user level languages for describing diagnosis knowledge but do not require detailed knowledge of formal logics. In fact the language AD2L was purposely defined to have an appearance similar to conventional programming languages that would help acceptance with engineers or software developers. We describe the principles of AD2L, and then discuss the implementation of the framework, its use in domains as diverse as circuit diagnosis and software debugging, and ongoing work on the system.

2 Model-Based Diagnosis

Model-based diagnosis (MBD) [28,5] is a general approach to solve the problem of diagnosing malfunctions in technical, biological, or environmental [17,16] systems. In MBD a declarative model SD of the system is used to identify components $COMP$ of the system that if assumed to be incorrect, cause the observed behavior OBS. Formally, diagnoses can be characterized in different ways, the most widespread being consistency-based [28]: A set $\Delta \subseteq COMP$ is a diagnosis iff

$$SD \cup OBS \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \cup \{AB(C) | C \in \Delta\}$$

is consistent. $AB(C)$ indicates that a component C is behaving abnormally, and a correctly behaving component C is described by $\neg ab(C)$. In general we want to compute diagnoses which are subset-minimal.

The model must be compositional, i.e., provide behaviors of individual components from which the overall system is composed (such that the system description can be composed from the models of the components) but requires only to capture the correct behavior. The faulty behavior of components can be also incorporated into the MBD framework (see [30,10]). The MBD approach is flexible and is not limited to diagnosis of physical systems, e.g., it has also been applied to solving configuration tasks [3,32] and software debugging [11,34].

The main task of a MBD system is to determine components that are responsible for a detected misbehavior. In consistency-based diagnosis this is done by assuming the correctness of components and proving consistency of the given model and observations. If the assumptions lead to an inconsistency, they are called a conflict. Reiter's hitting set algorithm [28,14] uses the conflicts to compute all minimal diagnoses. Hence, diagnosis is reduced to search for all conflicts. Beside [28] the GDE [5] makes use of this approach. Other MBD algorithms based on a form of belief revision [12] or on constraint satisfaction algorithms [7,31]. Most of the diagnosis algorithms utilize special data structures for search.

Apart from theoretical work on MBD and modeling for MBD there are multiple applications of MBD described in the literature. In [38,27] the authors describe a MBD system that operates the Deep Space One spacecraft. Other applications of MBD and model-based reasoning (MBR) are reported in [39,35]. For example, [25] introduces an MBR approach to nuclear fuel reprocessing, and [24,29,1] describe the application of MBD in the automotive domain, a very promising area to apply MBD technology.

3 Building MBD Applications: The Problem

An MBD application presupposes the existence of an implemented diagnosis engine and a model of the system that can be described using the language used by the diagnosis engine. The diagnosis engine makes use of the model and the given observations to compute (minimal) diagnoses. Most prototypical diagnosis systems tightly couple the diagnosis engine and the system description language which is used to describe the model. This has the advantage that there is no overhead on side of the modeling language, but has the disadvantage that models cannot be used by other diagnosis systems without substantial effort. What is required in order to solve this problem is a general system description language with well-founded syntax and semantics. Such a language must be capable of describing different kinds of systems from different domains.

Although the use of a standardized and general system description language has its advantages, a general diagnosis framework should avoid too tight a coupling. Reasons are: (1) languages change, (2) in some applications it is better to use the basic model representation methods directly, (3) a general framework should be easily adaptable to other circumstances, and finally (4) the implemented diagnosis engine may not be capable to handle all aspects of the language because it is optimized for a given subset. Therefore, it is better to introduce a compiler that maps models described in a modeling language to the basic model representation methods provided by the diagnosis engine. The compiler has to ensure not only syntactical correctness but also the correct mapping of models to their corresponding representation.

We propose the use of a general modeling language which allows for specifying not only the structure of a system and the behavioral models of the components but also additional diagnosis knowledge, e.g., fault probabilities, possible replacements and repair suggestions, observability of connections and states, correctness of components and component focus sets, logical rules stating physical impossibilities as described in [10], and others. Every diagnosis engine that is capable of compiling the models written in such a general modeling language can make use of them. If using the proposed approach, we gain more flexibility, enhance model reuse, and focus the user more on modeling issues than on implementation issues.

4 The DiKe Modeling Language and Implementation

Our MBD application framework comprises two main parts: a modeling language (AD2L) and class library implementing different diagnosis engines. The modeling language allows specifying the behavior of components and the structure of systems. It is independent from the implemented diagnosis classes and could be used in other systems. Syntax and semantics of AD2L are well-defined. We have used the DiKe application framework in several different diagnostic systems.

4.1 The AD2L Modeling Language

The purpose of designing a dedicated system description language for model-based diagnosis is to support the user in writing the actual models. He should not be required

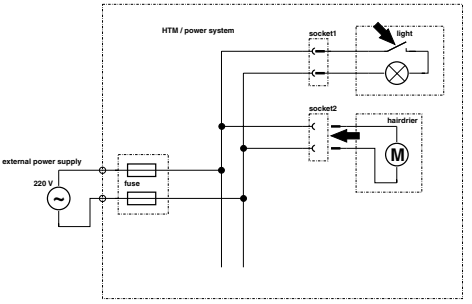


Fig. 1. Parts of a home power network

to engage in applications programming, and the language should provide constructs to directly express the basic primitives that are generally used in system descriptions for model-based diagnosis. In other words, the language is supposed to provide a vocabulary that corresponds to the structure generally present in system descriptions for various domains.

We assume a diagnosis model to be composed out of smaller *model fragments*. Such a model fragment describes the behavior of a single component, e.g., a n-input AND gate, whereas a complete model describes the structure and behavior of a whole system in a logical way. The art of writing model fragments is that of describing the behavior in a context independent way, i.e., the behavior description of a component should not determine its use. In practice context independence cannot always be achieved, nor is it possible to define a language that guarantees context independence.

In this section we introduce the basic concepts of the AD2L language [26] designed for the purpose of communicating diagnosis knowledge. Instead of formally describing the language we show its capabilities using an example from the electrical domain. Consider a home power network, which typically involves a connection to the local power supplier, fuses, sockets, and devices attached to sockets: lights, washers, and other power consumers. Figure 1 shows a small part of such a net.

In order to write a model for a power network we (1) define types for connections, (2) declare a model fragment for every component, and (3) connect the fragments to receive the final model.

Defining types. Types are used for representing the domain of connections and component ports. In AD2L there are 5 predefined types: boolean, character, string, integer, and real, with some predefined functions, e.g. +,*, and others for integer and real values. In addition, the programmer can declare enumeration types. For example, in the power network domain we want to describe a qualitative model for currents and voltages, only using the information about whether a current or voltage flows or not. In this case we define the following type:

type electrDomain **:** { “on“ **,** “off“ }**.**

Apart from such simple enumeration types, AD2L allows the use of predicates and the specification of tolerances and equivalences.

type quantDomain **: real tolerance [** -5% **,** 10% **].**
type myLogic **:** { '0' **,** 'L' **,** '1' **,** 'H' **,** 'X' **,** 'Z' }
equivalence { '0' **=** 'L' **,** '1' **=** 'H' }**.**

Tolerances and equivalences are used for determining a contradiction during computation. For example, if we can derive the value '1' for a connection S of type *myLogic* and we have an observation 'H' for S, then no contradiction arises. If no equivalence relations are defined, a contradiction occurs because it is assumed that a connection can only have one value.

The use of predicates in type declarations is another feature of AD2L. Consider the case where a connection can have several values, e.g., a radio link that broadcasts the signal of several channels at the same time. The type for this connection is defined as:

type channel **:** { "nbc" **,** "cnn" **,** "abc" }**.**
type radioLink **:** { **predicate** online **(** channel **)** }**.**

The *channel* type enumerates all possible channels that can be broadcasted. A contradiction only occurs in this case if a connection of type *radioLink* has a predicate and its negation as its value at the same time, e.g., *online("abc")* and *-online("abc")*.

Using types for connections has two advantages. The first is that type checking can be performed at compile time. The second is that the list of domain values can be employed at the user interface level to present a list of possible values, or for checking the validity of user input after data entry.

Writing behavior models. The *component declaration statement* is the basic tool in AD2L for describing the interface and behavior of components. AD2L distinguishes between two different component declarations, atomic components and hierarchical components. Atomic components have a fixed, declared behavior and cannot be subdivided further. Hierarchical components derive their behavior from their set of internal subcomponents (and connections between them) which are separately described. The subcomponents themselves may either be hierarchical components or atomic components.

Using the power net example, we now show the use of AD2L for writing atomic components. Verbally speaking, a light is on if its switch is on and it is connected to a current source. If the light is on, there must be a current flow and a voltage drop. Note that a voltage can be measured although there is no light and no current flowing through the bulb. If the bulb is broken, i.e., the component does not work as expected, then there is no current flow and the light is off. Formally, this behavior can be described in AD2L as follows:

component light
comment "This is a qualitative model of a light"
input current, voltage : electrDomain.
input switch_on : **bool**.
output light_on : **bool**.

```
    default behavior nab
      Val(switch_on,true), Val(voltage,on) =:= Val(current,on).
      Val(current,on) =:= Val(light_on,true).
      Val(light_on,false) =:= Val(current,off).
      Val(light_on,false) =:= Val(switch_on,false).
    end behavior

    behavior ab
      =: Val(current,off).
      =: Val(light_on,false).
    end behavior
end component
```

In the first line of the AD2L declaration of the component *light*, a comment is given. It is followed by the declaration of the interface, i.e., the *ports* which are used for connecting different components via *connections*. The AD2L compiler checks the types of connected ports and reports an error if they are not equivalent. In our case we define 4 ports: *current*, *voltage*, *switch_on*, *light_on*. The declaration of interfaces allows to specify whether a port is an input or output port or both (**inout**). Note that this information is not used to restrict the behavior description. It is intended to be used by diagnosis engines to determine a focus set or to optimize questions to the user about values. In addition, in AD2L the programmer can specify parameterizable *generic ports*. A generic port can be used to configure the component for different systems. For example, a component with a generic number of inputs is defined by:

```
generic Width : integer = 2.
input i[1-Width] : bool.
```

After the interface, the behavior of the component can be defined. It is possible to define several behaviors. Each of them has a name (also called a *mode*), e.g., *nab* standing for *not abnormal*. In the example we distinguish between two modes. One defines the expected and the other the faulty behavior of *light*. AD2L requires one mode to be designated as default mode. The default behavior is used by the diagnosis engine as a starting point for diagnosis.

A behavior itself is described using rules. A rule consists of two parts (the left and the right side) separated by an operator $=:$ or $=:=$. For rules of the form $L =: R$ the semantics are easy: If L evaluates to true, then all predicates in R must be true. Rules of the form $L =:= R$ are a shortcut for $L =: R$ and $R =: L$. For rules of the form $L =: R$ the left side is called *condition* and the right side *action part* (where the action simply consists of asserting the predicates on that side as true).

The left and the right side of rules are conjunctions of predicates. Disjunctive sentences have no direct representation in AD2L for complexity reasons. Predicates are predefined. The use of quantifiers is possible. Note that this AD2L predicates are different from data type predicates that are used as elements of a type and are defined by using the **predicate** keyword. Data type predicates are explained previously. The most important AD2L predicate is the **Val** predicate. Its first argument is the port and the second the value of the port. It evaluates to true if the port has the given value. Another

important predicate is **Cond** with a condition as the only argument. If the condition is true, the predicate evaluates to true. For example, the rule

Val(anInput,X), **Cond**(X>20) =: **Val**(anOutput,true).

specifies that if the value of *anInput* is greater than 20 the port *anOutput* must contain the value *true*. Note that **Cond** can only be used in the condition part of a rule. (Thus, in rules containing **Cond** the use of =*:*= is not allowed.) Another predicate is **Fail** which, if true, raises a contradiction. This predicate has no arguments and can only be used in the action part of a rule. Again, its use in =*:*= rules is not allowed.

The use of quantifiers in rules is defined in AD2L. The intention is to use quantifiers for making the model as concise as possible. For example a quantifier can be used in the case we have to set all input ports to a specific value.

=: **forall INPUTS** : **Val**(**INPUTS**,on).

Note, that the existential quantifier (**exists**) can only be used in the condition part. In this case only the =*:* rule operator is allowed. The **forall** can be used in both parts of the rule. The quantification operator only influences the part of the rule where it is used. All of these restrictions are necessary to avoid complexity problems.

The variable **INPUTS** is a built-in variable storing all input ports of the current component. There are several other built-in variables predefined in AD2L, e.g., **OUTPUT** and others. The user can also define variables using the **variable** declaration that must be located in the interface part of the component declaration. All variables are restricted to a finite domain.

We define the semantics of quantifiers based on the semantics of rules and predicates.

Forall Conjunctive sentences of the form ***forall*** *X: P(X) op A* (with $|X| = n$) are transformed into a single sentence *P(v_1), ..., P(v_n) op A*, where v_i is an element of X and *op* is either =*:* or =*:*=.

Exists Conjunctive sentences of the form ***exists*** *X: P(X) =: A* (with $|X| = n$) are transformed into a set of sentences *P(v_1) =: A*,..., *P(v_n) =: A* one for each element v_i of X.

The user can extend the core behavior definition by additional properties, i.e., repair costs, actions, and probabilities.

```
component light
...
    default behavior nab
      prob 0.999
      cost 2
      action "Replace the bulb"
      Val(switch_on,true), Val(voltage,on) =:=
        Val(current,on).
...
```

As stated above, hierarchical components can also be defined in AD2L. Their declaration is discussed in the next section. We decided not to distinguish between hierarchical components and systems because there is no conceptual difference between them - both contain components and connections.

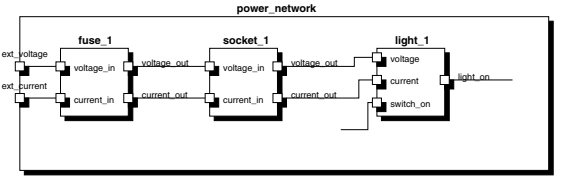


Fig. 2. The graphical representation of *power_network*

Writing system models. Systems and hierarchical components consist of components and connections. Components can be either atomic components or again hierarchical components. The behavior of a system and a hierarchical component is given by the behaviors of the subcomponents. A hierarchical component can only have two behaviors. If it works correctly, all subcomponents are assumed to work correctly as well. The subcomponent behavior is given by their default behavior. In the other case, where the hierarchical component is assumed to fail, nothing can be derived. The probability of a hierarchical component C working correctly is computed using the probabilities of the default modes of the subcomponents $\{C_1, \ldots, C_n\}$:

$$p(nab(C)) = \prod_{i=1}^{n} p(default\_mode(C_i)).$$

From the rules of probability theory follows $p(ab(C)) = 1 - p(nab(C))$.

The user defines systems and hierarchical components by (1) declaring the used subcomponents, and (2) defining the connections between them. In our example the power net can be described at the system level as follows:

component power_network
  input ext_voltage, ext_current : electrDomain.

  subcomponents
    fuse_1 : fuse.
    socket_1 : socket.
    light_1 : light
  end subcomponents

  connections
    ext_voltage -> fuse_1(voltage_in).
    ext_current -> fuse_1(current_in).
    fuse_1(voltage_out) -> socket_1(voltage_in).
    fuse_1(current_out) -> socket_1(current_in).
    socket_1(voltage_out) -> light_1(voltage).
    socket_1(current_out) -> light_1(current).
  end connections
end component

The graphical representation of the $power\_network$ system is given in figure 2.

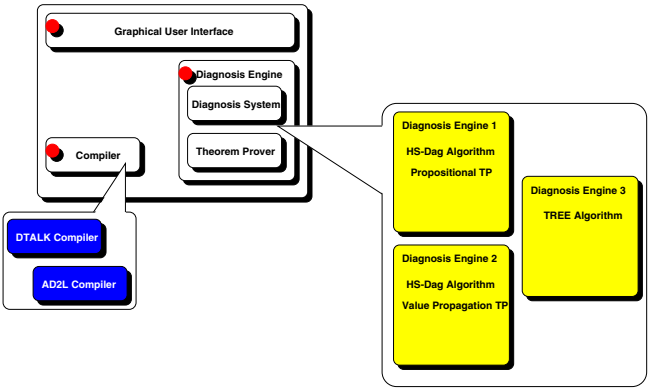


Fig. 3. The Diagnosis Kernel (DiKe)

4.2 The DiKe Framework Implementation

The diagnosis kernel implements all classes and methods necessary for building a diagnosis application, i.e., the class library for the user interface, the diagnosis engine, and the compiler. It was designed for flexibility and ease of use. The diagnosis kernel framework is implemented in Smalltalk (Visualworks 2.52 and 5i) and comprises generic classes for representing general interfaces and specific classes implementing the functionality. The portability of the Visualworks system has led to use of the framework under Solaris, Linux, and Win 95/98/NT. Figure 3 gives an overview of the currently implemented parts. The diagnosis engine on the right is divided into a diagnosis system and a theorem prover. The diagnosis system implements a diagnosis algorithm and stores knowledge about observations, connections, and components of a specific system. The theorem prover stores the behavior of the component to allow checking whether a system together with the observations and assumptions about the correctness of components is consistent or not. In cases where a consistency check is not necessary, a theorem prover is not used, e.g., the implementation of the TREE algorithm [31] requires no explicit theorem prover. The implementation of Reiter's hitting set algorithm [28,14] on the other hand needs a theorem prover.

Currently, our framework provides three different diagnosis engines. Two engines use Reiter's algorithm while the other implements the TREE algorithm. Although the diagnosis algorithm is the same for the first two implementations, they use different theorem provers. One uses a propositional theorem prover and the other a constraint system and value propagation. All concrete implementations have the same generic superclass. The generic diagnosis system class provides the interface, e.g., names of methods for executing diagnosis, requesting the next optimal measurement point, adding and removing observations, and others. The user writing an application using our diagnosis framework should choose the most appropriate diagnosis engine. If the model contains operations on numbers, the user should choose the value propagation algorithm. If the model is tree structured as defined in [31] the user should take the TREE algorithm. In all

other cases the algorithm using the propositional theorem prover ensures best runtime performance that is almost equal and sometimes better than the performance published for other algorithms [12,37].

The diagnosis kernel provides two languages for describing specific diagnosis systems, e.g., a digital full-adder. The first language, DTalk is closely related to Smalltalk syntax and semantics. For every kind of diagnosis engine there are specific language constructs representing the distinct behavior descriptions. While the knowledge about structural properties of a diagnosis systems are almost the same for every engine, this is not the case for the component models of DTalk. Therefore, we have developed a second more general language. This language *AD2L* has been described in a previous section. Models written in AD2L are not restricted to one diagnosis engine, although currently only the transformation of AD2L programs into the representation for the constraint based diagnosis engine is supported.

Apart from classes for representing diagnosis knowledge, we have added classes for building user interfaces to the diagnosis kernel, to enable rapid prototyping of complete diagnosis applications. Using the demo applications and the diagnosis kernel classes as starting point, a first prototype of a diagnosis system implementing most of the required diagnosis functionality can be developed quickly. One of the demo interfaces uses a text-based user interface allowing to load systems and handle observations and other diagnosis knowledge, e.g., fault probabilities. The second variant uses a graphical approach for representing components and connections, similar to a schematics editor. Both applications provide messaging interfaces for starting the diagnosis and measurement selection process.

Diagnosis and measurement selection runtimes are competitive with other implementations [12,37,7]. Parts of our VHDL debugger [11,33,40] were implemented using the diagnosis kernel.

5 JADE: A Debugger for Java Programs

The DiKe class library has been used for several MBD projects. One of the most recent projects using the DiKe library is the *Java Diagnosis Experiments* (JADE) project. During this project the MBD framework is used to implement a debugger for Java programs. We have developed two different models of Java programs. One abstract model [21,20] considers only the dependencies between variable occurrences in the program which are stored as propositional rules. The other model [23] represents the whole semantics of a (large) Java subset. This subset includes method calls, conditional statements, and while statements. This value-based model is represented as a constraint propagation system. Because of the different representations the implementation of the models makes use of different diagnosis engines. The abstract model is mapped to classes implementing the propositional theorem prover, whereas the value-based model is mapped to the implemented constraint propagation system. Both model implementations make use of the implemented hitting set diagnosis algorithm.

The JADE debugger is a prototype system for research purposes and for demonstrating the underlying model-based techniques. Development of the debugger was significantly accelerated by making use of the available DiKe framework. First, no changes of

the basic classes of the DiKe library were necessary, we only needed to develop classes implementing the models. Because of available classes implementing similar functionalities and inheritance this was not a problem. Second, the standardized interface of the different diagnosis engines makes it easier to develop a graphical user-interface. Only small changes were necessary to adapt the interface of the dependency-based model to use it as an interface for the value-based model. Finally, the DiKe class library is very stable, because it has been tested on a number of examples and has been used for several prototypes so far. Because of the use of the DiKe framework the first Jade prototype could be finalized early in the project. The most expensive part for realizing the first prototype was the implementation of a Java compiler, the Jade interface, and the development of the models. As a consequence we were able to extend the debugger to support the whole Java language and to improve the user-interface which is very important.

Other prototypes where we make use of the DiKe class library are a debugger for the hardware design language VHDL [11], a system allowing to interchange component models using TCP/IP socket communication, and a reconfiguration system for software parameters of a phone switching system [32], all in the context of industrial projects.

6 Results

The DiKe MBD framework has been used to build prototypes for several different domains, e.g., debugging of VHDL designs [11], reconfiguration of software parameters of phone switching systems [32], audio routing systems, and more recently debugging of Java programs [22,23]. In all of these prototype applications the framework has been proven to be flexible enough and complete with respect to the provided functionality. The expressiveness of AD2L has been tested on several example systems.

Besides providing a well designed framework for MBD applications, the improvement of diagnosis algorithms was also a goal of several projects in the past years. TREE and more recently TREE* is one of the outcomes of the projects that were integrated into the framework. In the following we compare the TREE* algorithm which is an extended version of the TREE algorithm, with El Fatah and Dechter's SAB diagnosis algorithm [7]. Figure 4 gives the runtime results of TREE, TREE*, and SAB for tree-structured digital systems comprising *And* and *Or* gates as described in [7]. We see that both TREE and TREE* outperform SAB which was proven by [7] to be faster than GDE [5] and Reiter's algorithm [28]. This holds especially for larger systems where a short runtime becomes an major issue. In [36] TREE and TREE* are described in detail and more empiricial results are given.

7 Related and Future Work

Since the beginnings of model-based reasoning several techniques for representing models have been proposed. They mainly have in common that they are qualitative in nature, i.e., they do not use quantitative values. Such models are not only used in MBD but also in other fields. For example hardware designers speak about "low" and "high" or "true" and "false" instead of the exact voltage levels. In [4] an overview of qualitative modeling is given. Although the basic modeling principles seem to be established, there is almost

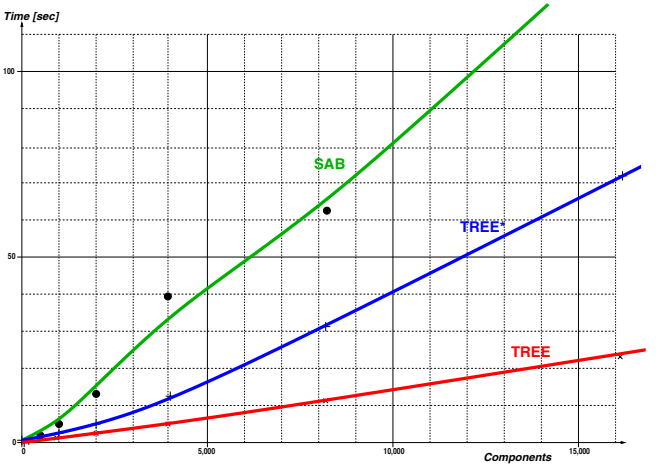


Fig. 4. Comparing TREE, TREE*, and SAB

no accepted and widely used model description language available. Every reasoning system based on specific models uses its own languages. In addition, apart from [18], where a WWW-based modeling system for sharing knowledge about physical systems is described, almost no work in the direction of providing tools for handling models and model libraries has been done. This system uses CML (Compositional Modeling Language) for describing models that can be translated to the Knowledge Interchange Format (KIF) [13]. CML combines languages used for describing systems using Qualitative Process Theory [9] and the Qualitative Physics Compiler [2]. Other approaches for sharing diagnosis knowledge include [15] where KQML [8] is used as communication language. Recent approaches for model interchange are mostly based on XML. We did not take this approach, because we consider XML to be primarily a language for information exchange, which does not provide support for defining semantics specific to modeling for diagnosis. On the other hand, it is straightforward to convert AD2L to an XML-based format.

All previous approaches that rely on a logical description of the model are well suited for representation purposes. However, they are not so good when modeling is to be done by less experienced users. We face this problem in industry, where people are not familiar with the concepts of MBD and logic description languages (including Prolog). Although they see advantages in MBD compared to other approaches, they are sceptical concerning the realization of the advantages, e.g., reuse of models. Teaching students (especially from the electrical and mechanical engineering fields) the fundamentals of model-based diagnosis might alleviate the problem in the long term. A major step forward on the road to more general acceptance could be to uncouple the representation issue from the theoretical roots of the field and provide a dedicated representation that is more in line with the background of practitioners who might be "put off" by the appearance of pure logic. Advantages of a widely accepted language would include the possibility to

interchange models between researchers and companies, or between companies directly, the increase of reuse, and the certainty for companies that the model description can be used for a long time, thus saving the investments for modeling and providing an argument for using MBD.

The language AD2L described in this paper is a proposal for such a modeling language. AD2L [26] has been developed as part of a project with the goal of interchanging system descriptions over the Internet, and has been extended and adapted for industrial needs afterwards. The language definition is independent of the underlying diagnosis engine and provides language constructs directly representing model-based concepts, e.g., components and connections. In addition, other concepts from programming language design have been incorporated such as packages and strong typing. This allows for building model-libraries and avoids errors at runtime, which are central requirements of industry.

Similar approaches have been considered in the past, such as the language COMODEL [6], but have not found general use in industrial applications. AD2L on the other hand was developed in collaboration with industry.

Although the introduced DiKe framework has been successfully used to implement different MBD prototype applications there are some open issues to be addressed. First, the class library contains different classes implementing the diagnosis engines and the AD2L language compiler. In our current implementation the AD2L programs are compiled to a structure that can only be used by one diagnosis engine. A future implementation should make a decision about which diagnosis engine to be used in order to optimize the overall diagnosis runtime. For example, a system that is tree-structured should be diagnosed using the engine implementing the TREE algorithm [31].

8 Conclusion

In this paper we have described an implementation framework for model-based diagnosis systems that we have used in the last three years to implement systems as diverse as classical circuit diagnosis, reconfiguration of telecommunication networks, and a knowledge-based software debugger. The framework provides a graphical user interface, different diagnosis engines with different computational properties so that system performance can be adapted to the requirements and properties of a particular domain. It includes two different modeling languages, of which one, AD2L was specifically designed to provide a system independent platform for diagnosis knowledge base development and to be amenable to non-AI developers and engineers.

References

1. Fulvio Cascio, Luca Console, Marcella Guagliumi, Massimo Osella, Andrea Panati, Sara Sottano, and Daniele Theseider Dupré. Generating on-board diagnostics of dynamic automotive systems based on qualitative models. *AI Communications*, 12(1/2), 1999.
2. James Crawford, Adam Farquhar, and Benjamin Kuipers. QPC: A compiler from physical models into qualitative differential equations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 365–372, Boston, August 1990. Morgan Kaufmann.

3. Judith Crow and John Rushby. Model-based reconfiguration: Toward an integration with diagnosis. In *Proceedings AAAI*, pages 836–841, Los Angeles, July 1991. Morgan Kaufmann.
4. Philippe Dague. Qualitative Reasoning: A Survey of Techniques and Applications. *AI Communications*, 8(3/4):119–192, 1995.
5. Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
6. Werner Dilger and Jörg Kippe. COMODEL: A language for the representation of technical knowledge. In *Proceedings* 9^{th} *International Joint Conf. on Artificial Intelligence*, pages 353–358, Los Angeles, CA, August 1985. Morgan Kaufmann.
7. Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *Proceedings* 14^{th} *International Joint Conf. on Artificial Intelligence*, pages 1742 – 1748, 1995.
8. Tim Finin, Yannis Labrou, and James Mayfield. KQML as an Agent Communication Language. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 291–317. AAAI Press / The MIT Press, 1997.
9. Kenneth D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.
10. Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Physical impossibility instead of fault models. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 331–336, Boston, August 1990. Also appears in Readings in Model-Based Diagnosis (Morgan Kaufmann, 1992).
11. Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.
12. Peter Fröhlich and Wolfgang Nejdl. A Static Model-Based Engine for Model-Based Reasoning. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, Nagoya, Japan, August 1997.
13. M.R. Genesereth and R.E. Fikes. Knowledge Interchange Format, Version 3.0, Reference Manual. Technical report, Technical report Logic-92-1, Stanford University Logic Group, 1992.
14. Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.
15. Florentin Heck, Thomas Laengle, and Heinz Woern. A Multi-Agent Based Monitoring and Diagnosis System for Industrial Components. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, pages 63–69, Cape Cod, Massachusetts, USA, May 1998.
16. Ulrich Heller and Peter Struss. Transformation of Qualitative Dynamic Models – Application in Hydro-Ecology. In *Proceedings of the* 10^{th} *International Workshop on Qualitative Reasoning*, pages 83–92. AAAI Press, 1996.
17. Ulrich Heller and Peter Struss. Conceptual Modeling in the Environmental Domain. In *Proceedings of the* 15^{th} *IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 6, pages 147–152, Berlin, Germany, 1997.
18. Yumi Iwasaki, Adam Farquhar, Richard Fikes, and James Rice. A Web-Based Compositional Modeling System for Sharing of Pysical Knowledge. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, 1997.
19. Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. AI Support for Debugging Java Programs. In *3rd Workshop on Intelligent SW Eng.*, Limerick, Ireland, June 2000.
20. Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. JADE - AI Support for Debugging Java Programs. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligence*, Canada, November 2000. Also appears in [19].
21. Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java programs using a model-based approach. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, Scotland, 1999.

22. Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Locating bugs in Java programs – first results of the Java Diagnosis Experiments (Jade) project. In *Proceedings IEA/AIE*, New Orleans, 2000. Springer-Verlag.
23. Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling Java Programs for Diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany, August 2000.
24. Heiko Milde, Thomas Guckenbiehl, Andreas Malik, Bernd Neumann, and Peter Struss. Integrating Model-based Diagnosis Techniques into Current Work Processes – Three Case Studies from the INDIA Project. *AI Communications*, 13, 2000. Special Issue on Industrial Applications of Model-Based Reasoning.
25. Jacky Montmain. Supervision Applied to Nuclear Fuel Reprocessing. *AI Communications*, 13, 2000. Special Issue on Industrial Applications of Model-Based Reasoning.
26. Christian Piccardi. AD^{2}L An Abstract Modelling Language for Diagnosis Systems. Master's thesis, TU Vienna, 1998.
27. Kanna Rajan, Douglas Bernard, Gregory Dorais, Edward Gamble, Bob Kanefsky, James Kurien, William Millar, Nicola Muscettola, Pandurang Nayak, Nicolas Rouquette, Benjamin Smith, William Taylor, and Ye-wen Tung. Remote Agent: An Autonomous Control System for the New Millennium. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany, August 2000.
28. Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
29. Martin Sachenbacher, Peter Struss, and Claes M. Carlén. A Prototype for Model-based Onboard Diagnosis of Automotive Systems. *AI Communications*, 13, 2000. Special Issue on Industrial Applications of Model-Based Reasoning.
30. Peter Struss and Oskar Dressler. Physical negation — Integrating fault models into the general diagnostic engine. In *Proceedings* 11^{th} *International Joint Conf. on Artificial Intelligence*, pages 1318–1323, Detroit, August 1989.
31. Markus Stumptner and Franz Wotawa. Diagnosing Tree-Structured Systems. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, Nagoya, Japan, 1997.
32. Markus Stumptner and Franz Wotawa. Model-based reconfiguration. In *Proceedings Artificial Intelligence in Design*, Lisbon, Portugal, 1998.
33. Markus Stumptner and Franz Wotawa. VHDLDIAG+:Value-level Diagnosis of VHDL Programs. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, Cape Cod, May 1998.
34. Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, pages 1074–1079, Stockholm, Sweden, August 1999.
35. Markus Stumptner and Franz Wotawa, editors. *AI Communications, Special Issue on Industrial Applications of Model-Based Reasoning*, volume 13(1). IOS Press, 2000.
36. Markus Stumptner and Franz Wotawa, editors. Diagnosing Tree-Structured Systems. *Artificial Intelligence*, volume 127(1):1–29. Elsevier, 2001.
37. Brian C. Williams and P. Pandurang Nayak. A Model-based Approach to Reactive Self-Configuring Systems. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, pages 267–274, 1996.
38. Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, pages 1178–1185, 1997.
39. Franz Wotawa, editor. *AI Communications, Special Issue on Model-Based Reasoning*, volume 12(1/2). IOS Press, 1999.
40. Franz Wotawa. Debugging synthesizeable VHDL Programs. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, 1999.

Constraints Applied to Configurations

Gerhard Fleischanderl

Siemens AG Österreich, Program and System Engineering,
A-1030 Vienna, Austria
Gerhard.Fleischanderl@siemens.at

Abstract. A configurator using generative constraint satisfaction is presented. The configurator is applied for large EWSD telecommunication switches produced by Siemens AG. The configurator has been in production use for more than four years and has required remarkably little maintenance effort. The knowledge representation with declarative constraints proved successful for modeling and solving large configuration tasks for real-world systems.

1 Introduction

Configuration is a challenging application domain for Artificial Intelligence techniques. The development of configurators is demanding for several reasons:

- The task requires knowledge of experts.
- The configuration requirements change frequently, leading to frequent changes in the component library and configuration constraints.
- Configurator development time and maintenance time usually are short.
- Users in different departments - such as development, sales, manufacturing, and service - often have conflicting goals as well as different automation needs, e.g. full automation of the configuration process.

We show how we mastered those challenges with constraint satisfaction.

2 Configuration Process

The general configuration process with a knowledge-based configurator is shown in Figure 1. In [1] an overview of AI techniques for configuration is presented.

In our application projects we use generative constraint satisfaction [2]. This AI technique proved suitable for mastering the challenges of large real-world configuration tasks. Generative constraint satisfaction employs a constraint network for problem solving which is extended during the configuration process. Generative constraints

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 455–458, 2001.

hold for all components of a given type and are used as generators for extending the configuration with new components and connections.

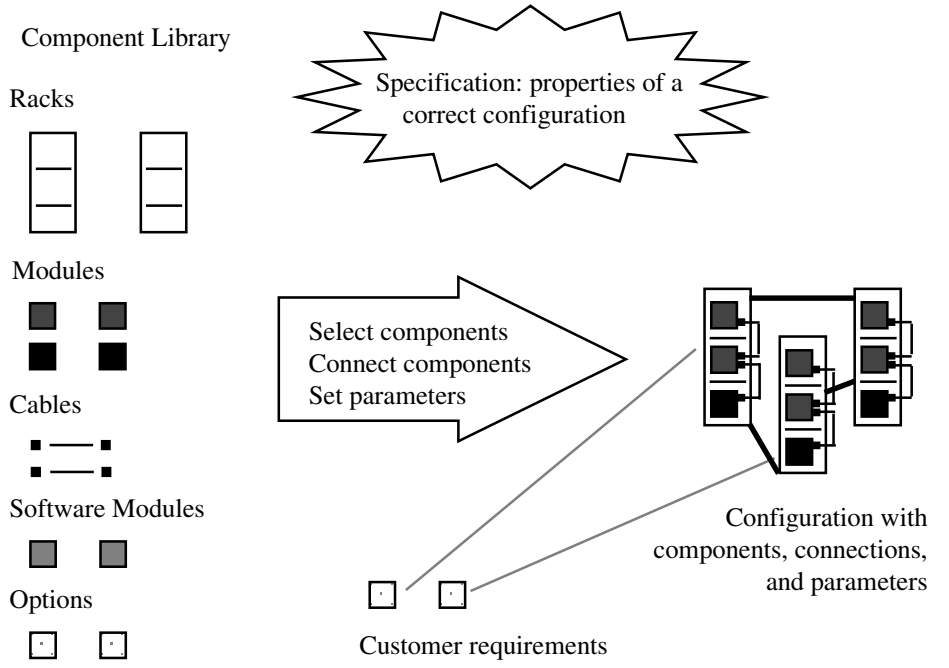


Fig. 1. The configuration process uses a *component library*, a *specification* (set of constraints) and the *customer requirements* to compute a configuration of connected and parameterized components. Valid configurations are specified by constraints which guide the configurator in selecting components from the component library

3 Modeling and Constraint Satisfaction

Our constraint-based configurator needs only few types because properties can be inherited. During configuration the parameters can be adjusted to configure the modules themselves. Consequently, knowledge bases for our configurator are smaller than for other problem-solving methods and can be maintained more easily.

Valid configurations are specified with configuration constraints, for example

- The first slot of every frame is reserved for power-supply modules.
- The amount of electrical power consumed by all power-consuming modules in a frame must be less than or equal to the power output of the corresponding power-supply module.
- The width of each frame must be less than 770 mm.
- Analog and digital modules must not be mixed in a frame.

The capabilities of the underlying knowledge-representation scheme are crucial for developing and maintaining the configuration, as well as for implementing configuration-process tasks such as validating consistency and generating meaningful explanations. Consequently, our goal was to develop a knowledge-representation scheme general enough to be useful in a variety of application domains.

Knowledge base. For configuration domains, component libraries describe the component types that can be used to build a configuration. We organize *component types* in an inheritance hierarchy that can be exploited during configuration.

The constraint language for the knowledge bases of our configurator lets users express all types of functionality, from that of key components to the formulation of complex customer-specified constraints.

An *attribute* declaration consists of the attribute's name and type (such as Number, String, Enum, or Boolean), and, optionally, an initial (and backtrackable) or a constant value. During configuration, users must correctly set attributes for each component. They can do this manually or automatically using the configuration engine.

Ports establish connections between two components. Each component type can have a set of port declarations, which consist of the port's name, type, and domain.

Constraints are a natural way to specify a relationship between attribute values or component ports. Constraints are always defined locally at a single component type. They can reference neighbor components, thereby navigating through the configuration. The declarative constraints also help users maintain the knowledge base:

- The same constraint can be used to generate or check a configuration.
- Constraints offer powerful representation and reasoning capabilities that can be extended to naturally express more complex configuration knowledge.
- The simplicity and declarative nature of the basic constraint model let users define knowledge bases with precise semantics.

4 Application

Our configurator performs all core tasks in the configuration process. It automatically generates and expands the system, it modifies, checks, and explains the configuration. Users can manipulate the configurations interactively, regardless of whether configurations were just loaded or were generated by the configurator.

Configuration outputs are primarily used to produce various part lists, assembly plans, and files for order processing and EWSD systems installation, including the cable plans. The configuration and reusable components are stored to serve as inputs for later use, such as existing-system expansion. Consequently, the configurator supports many stages of the product life cycle, including sales, engineering, manufacturing, assembly, and maintenance.

Configurator for telecommunication. We have successfully applied generative constraint satisfaction for configuring the EWSD (Elektronisches Wählsystem) digital switching systems developed and manufactured by Siemens AG. We implemented it [3] using our domain-independent configuration tool, COCOS (configuration by constraint satisfaction).

With the configurator the sales engineers produce large EWSD configurations that consist of approximately 200 racks, 1,000 frames, 30,000 modules, 10,000 cables, and 2,000 other units each. The knowledge base comprises approximately 20 types of racks, 50 types of frames, 200 types of modules, 50 types of cables, and 50 types of functional units for controlling the switching system.

Concerning development costs and functionality of use, e.g. flexibility for the user during the configuration session, we compared our new configurator with two earlier configurators. The old configurators had been developed with non-AI software development methods and tools.

Numerous changes to the EWSD system triggered a redesign of the program code of the old configurators, resulting in significant maintenance effort. In our new knowledge-based configurator minor changes to the knowledge base could cover all these requests. In addition, the knowledge base is more compact than traditional software code. Changes in the product library that trigger a change at multiple points in the old configurators cause only a single point of change in our new configurator.

Maintenance costs over several years depend on the technology of the configurator kernel and on the user interface design. During 4 years of its production use the maintenance effort per year of our configurator for EWSD switches amounted to between 5% and 10% of the initial development effort, which is very efficient.

5 Conclusion

Our application of generative constraint-satisfaction techniques in the configurator for EWSD switches shows that expressive knowledge-representation languages can be successfully used in a production environment for large, complex domains. Using our techniques proved successful in terms of cost and time. It also enhanced functional capabilities. Such knowledge representation can contribute greatly to improve the maintenance of knowledge-based systems.

References

1. Sabin, D., Weigel, R.: Product Configuration Frameworks - a survey. IEEE Intelligent Systems & their applications, Vol. 13. No. 4. IEEE (1998) 42-49
2. Stumptner, M., Friedrich, G. E., Haselböck, A.: Generative constraint-based configuration of large technical systems. AI EDAM, Vol. 12, No. 4. Publisher(1998) 307-320
3. Fleischanderl, G., Friedrich, G. E., Haselböck, A., Schreiner, H., Stumptner, M.: Configuring large systems using generative constraint satisfaction. IEEE Intelligent Systems & their applications, Vol. 13. No. 4. IEEE (1998) 59-68

From Theory to Practice: AI Planning for High Performance Elevator Control
(Extended Abstract)

Jana Koehler

Schindler Aufzüge AG
current address: IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland
koe@zurich.ibm.com, http://www.informatik.uni-freiburg.de/~koehler

Abstract. Offering an individually tailored service to passengers while maintaining a high transportation capacity of an elevator group is an upcoming challenge in the elevator business, which cannot be met by software methods traditionally used in this industry.
AI planning offers a novel solution to these control problems: (1) by synthesizing the optimal control for any situation occurring in a building based on fast search algorithms, (2) by implementing a domain model, which allows to easily add new features to the control software.
By embedding the planner into a multi-agent system, real-time interleaved planning and execution is implemented and results in a high-performing, self-adaptive, and modular control software.

1 Customization and Efficiency Challenge Innovation

As many other industries, elevator companies are facing two main challenges today: (1) the pressure on building costs requires to improve the transportation capacity of elevator installations, (2) increasing competition challenges diversification and mass customization strategies, which aim at providing new and individually tailored services to passengers.

Schindler Lifts Ltd. has developed the Miconic-10™ elevator *destination control* where passengers input their destination before they enter the elevator. The Miconic-10™ control has been introduced into the market in 1996 and since then more than 100 installations have been sold worldwide. A 10-digit keypad outside the elevators allows passengers to enter the floor they want to travel to. After input of the destination, the control system determines the best available cabin and the terminal displays, which elevator the passenger should take. The current Miconic-10™ control uses a rule-based, heuristic allocation algorithm. It groups passengers with identical destinations together and thereby allows them to reach their destination faster and more comfortable.

By using identification devices such as smart cards, pin codes, or WAP phones, passengers can even be recognized at an individual basis and more individually tailored services can be implemented: *access restrictions* of certain passenger groups to certain zones in the building, *VIP service* to transport some

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 459–462, 2001.

passengers faster than others, e.g., medical emergency personal, and the *separation of passenger groups* that should not travel together in the same elevator.

Today, elevator systems can offer these services only in a very limited way by permanently or temporarily restricting the use of selected cabins. It is impossible to integrate and embed these functionalities directly into the usual normal operation of an elevator group and the restricted usage of some of the cabins dramatically reduces the transportation performance.

The work summarized in this abstract has been driven by a rigorous formal approach. In the early studies in 1998 and 1999, the complexity of the problem has been investigated and it has been proven that even in the simplest case, destination control is NP-hard [4]. Based on these results, a comparative analysis has been conducted, which modeled destination control as a *planning* problem, as a *scheduling* problem, and as a *constraint satisfaction* problem [3,2]. Modeling the problem from a planning perspective seemed to be the most natural approach [1] and resulted in the *miconic10* domain, which has also been used in the AIPS-00 planning systems competition. The formal modeling of relevant domain properties in PDDL allowed to precisely define the services and prove the domain-specific algorithm to be sound and complete. It also helped in generating test problems that were used to verify the implementation.

There are two aspects of the problem: The *static, offline optimization problem* for one elevator, which requires to compute an optimal sequence of stops for a given, fixed traffic situation in a building. The *dynamic, online decision problem*, which needs to cope with the immediate and unknown changes of traffic situations.

2 The Offline Problem: A Case for AI Planning

The offline optimization problem for one elevator is given by a particular traffic situation in a building: the state and position of the elevator, the unanswered destination calls of passengers who are waiting in the building, and the destination calls that have already been picked up and where passengers are currently traveling in the elevator towards their destination. Given a number n of destination calls with board floor b and exit floor e as $(b_1, e_1), (b_2, e_2), (b_3, e_3), \ldots, (b_n, e_n)$ we need to compute a totally ordered sequence of stops $s_1, s_2, \ldots, s_k$ such that each s_i corresponds to a given board or exit floor (no unnecessary stops should be contained in the sequence) and where each b_i is ordered before each e_i (since passengers have to be picked up first and then delivered to their destination).

In practice, one is interested in finding stop sequences that minimize a given optimization criteria, e.g., minimizing the average waiting times of passengers or minimizing the overall time the passengers spend with the elevator from the moment they insert the call until they reach their destination.

The domain specific planning algorithm constructs a sequence of stops for a single elevator out of the stop actions that are applicable in a given situation. The following properties turned out to be key to success: (1) the state representation and the choice of data structures to implement it, (2) the search algorithm

as a combination of a depth-first, branch-and-bound search with forward checking techniques from constraint reasoning, (3) an admissible heuristic function that estimates the distance to the goal state in such an effective way that the branching factor in the search space is reduced by 60 to 90 %. The planning system allows to find optimal plans of up to a length of 15 to 25 stops in less than 100 milliseconds even in search spaces containing between 10^{12} and 10^{15} states, which are frequently generated by data from real buildings with high traffic peaks. The planning system is able to deliver optimal plans given the tight real-time requirements because it works on the level of destination commands and returns an abstract sequence of stops. The execution of these plans requires to translate the stop sequences into the much more fine-grained level of elevator control commands.

3 The Online Problem: Real-Time Interleaved Planning and Execution with Failure Recovery Mechanisms

For each elevator, a so-called *jobmanager* has been developed, which controls a single cabin with its drive and various doors. A jobmanager is a holon of agents responsible for different tasks in the control, see Fig. 1. The agents communicate via an asynchronous messaging system supporting publish/subscribe mechanisms and allowing a peer-to-peer communication between the control and hardware components such as drives, doors, and terminals.

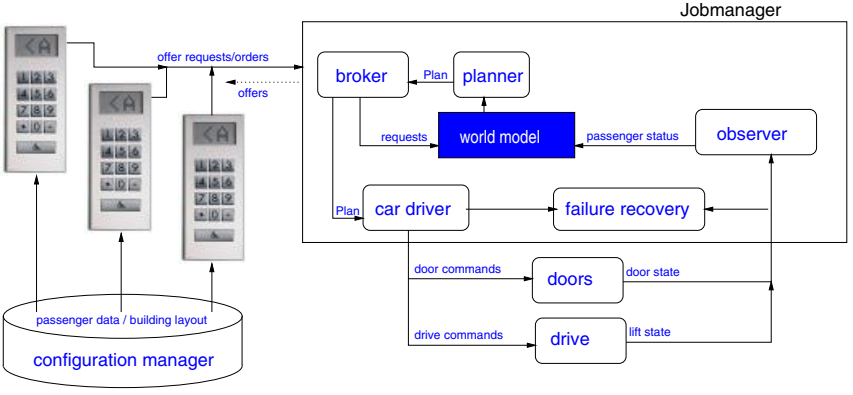


Fig. 1. The jobmanager as a multi-agent system implementing interleaved planning and execution for a single elevator.

The **broker** receives the offer requests from the terminals and adds the new calls to the world model, which is part of the initial state representation for the planner. It initiates the planning process and evaluates the returned plan. Based on the evaluation, it sends offers to the requesting terminals, which book passengers via an *auctioning* process. The **car driver** is responsible for executing

the plans. Given an abstract sequence of stops, the car driver maps these into a fine-grained temporal sequence of activities, e.g., accelerating, moving, landing, opening doors, etc. The **observer** updates the world model of the planner solely based on information that it receives from the doors and the drive. This model updating independently of the car driver's actions is very important to keep world model and reality in accordance. The **failure recovery** maps the activities of the car driver to the information it receives from drive and doors and verifies whether the planned activities are correctly executed. It implements a very flexible approach to deal with hardware failures, situations that made world model and reality fall apart, and passengers who behave not as assumed. The **configuration manager** provides information about the building layout, i.e., the number of floors, access zones, passenger groups and access rights, active services, etc.

Each component is a self-acting agent that initiates activities when certain events occur. This can trigger several agents simultaneously and their activities can run in parallel. The distributed control is able to deal with such interfering events.

The developed control software has achieved two major results: First, we obtain a much more modular and compact code comprising only 8000 lines in an object-oriented language. The underlying agent-based architecture leads to very clear interfaces and allows it to further develop the different agents independently of each other. The planning algorithm is able to unify the software for various elevator platforms, e.g., cabins with multiple decks.

The situation-dependent optimization improves the transportation capacity of elevator systems significantly. The currently used heuristic allocation algorithm already improved performance by 50 to 80 % when compared to conventional control systems where passengers press the destination buttons inside the cabin after they have been picked up. With the intelligent planning system, performance is increased by another 10 to 30 % depending on the traffic pattern. Average waiting times remain roughly identical, but travel times within a cabin get reduced by up to 35 %.

References

1. J. Koehler and K. Schuster. Elevator control as a planning problem. In S. Chien, S. Kambhampati, and C. Knoblock, editors, *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling*, pages 331–338. AAAI Press, Menlo Park, 2000.
2. Jana Koehler. Von der Theorie zur Praxis: Verkehrsplanung für Hochleistungsaufzüge. *Künstliche Intelligenz*, 2, 2001.
3. B. Seckinger. Synthese von Aufzugssteuerungen mit Hilfe von Constraintbasierten Suchverfahren. Master's thesis, Albert-Ludwigs-Universität Freiburg, 1999.
4. Bernhard Seckinger and Jana Koehler. Online-Synthese von Aufzugssteuerungen als Planungsproblem. In *13. Workshop Planen und Konfigurieren, Interner Bericht des Instituts für Informatik der Universität Würzburg*, pages 127–134, 1999.

Semantic Networks in a Knowledge Management Portal

Kai Lebeth

Dresdner Bank
Kai.Lebeth@Dresdner-Bank.com

Abstract. We present a navigable semantic network, that can be calculated automatically from document database, as an important component of a knowledge management portal that was developed for a corporate intranet. Users of the portal can navigate through a semantic network of terms, that helps users to determine the company specific meaning of terms. The semantic network utilizes the standard search engine of the portal and offers a query expansion mechanism for unknown terms, that retrieves only documents with a company specific default meaning.

1 Introduction

The terminology found in corporate intranets forms a mini-universum, which might differ significantly to an everyday life usage of that words and terms. In corporate intranets, one finds very specific meanings of words, that these words typically do not show in a non specialized context. For example in banking industry the term „Risiko“ (risk) is used in a lot of very specific ways, that comprise a variety of bank specific kinds of risks. On the other hand, there are terms that have a rather well defined default meaning in an every day life context that are not likely to be found with this non-specific meaning in the document-base of a specialized industry. In addition, a company’s terminology contains meanings for words, that are completely different to the meaning these words have normally. In Dresdner Bank, e.g., there is a very important application called „Bingo“, which has nothing to do with the favorite game. Typically, in corporate terminology, there exist a big group of terms, whose meanings are only known to specialists.

This situation creates a very difficult situation for gathering information from corporate documents knowledge bases like, e.g., a corporate intranet, when a user just sends a single term without additional information to a search engine. Unfortunately, the lack of additional information, is in a lot of cases the very reason for questioning a search engine.

Thus, satisfying the information needs of an employee depends on finding a specific meaning of rather general terms, or a company’s specific meaning of seldom terms, which are unknown to the user.

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 463–466, 2001.

In addition, gathering information in a business context means that it is important to identify the business process or the process owner that are correlated with a term. The identification of proper names of people that are responsible for a process or a product is often the most relevant result for the information need that is behind a user's query.

Standard search engines often can not meet this requirements, since they deliver whole documents and can not identify correlations between terms, proper names and names for processes, which are crucial for the understanding of a company's organization.

In this paper, we describe a component of a knowledge management portal for the corporate intranet of the Dresdner Bank. The tool visualizes term in the neighborhood of their most salient contexts. The relations between terms are displayed in a semantic net structure, in which the user can navigate through the company's individual conceptual structure of terms, people and processes.

2 The Architecture

The intranet of the Dresdner Bank contains more than 400.000 documents, most of them HTML-pages, documents in Microsoft Word format, Microsoft PowerPoint presentations and documents in PDF format. There is a standard search engine available that indexes these documents and offer standard search functionalities. In the design phase of the knowledge management portal, we decided on three main topics.

First the portal's information offering was designed on the base of an empirical model of the users' information needs. We used a web-data mining approach, where we analyzed the users' questions to the existing search engine and cluster them in groups of common information needs [1].

Secondly, we decided not to develop a new search engine for the retrieval of documents, but use the existing search engine on the top of a query-expansion mechanism feed by the semantic net component.

For the storage of the analytical data, we decided to use a standard relational database, that forms what we called a universal linguistic index (ULI) for the data. Performance optimizations and interfaces could be programmed using the database API, which guarantees professional support and a standard connection to the banks IT-infrastructure.

The systems consists of three modules: A data gathering component (a so-called „spider") that crawls the intranet, filters the different document formats and normalizes the documents by tokenization and a sentence-based segmentation. The second module is a linguistic analysis component that extracts named entities and

identifies syntactically licensed relations between terms. Using a simple anaphora resolution, the system is able to find semantic relations between terms that occur in different sentences by a simple coindexing mechanism.

The third component is a relational database that stores the output of the flat linguistic analysis the word and relation counts together with a document's meta-information like, e.g., the author of the document and its creation date.

3 Automated Compilation of Term Correlation Graphs

Big corpora contain a lot of implicit knowledge about the meaning of words in terms of the contexts a word typically occurs with. Knowing the most important contexts in which an unknown term is used, often yields enough information to determine the meaning of the word itself. Correlation analysis of words have quite a tradition in statistic computational linguistics and are often utilized in advanced information retrieval, e.g. [3]. Typically, a correlation measure between two words is calculated by counting the two words' overall occurrences and the number of common occurrences of both words in a document, a paragraph or a sentence. If two words tend to occur significantly more often than by chance, they can be assigned a positive correlation measure. Measures that have been proposed are chi^2, mutual information and others [4].

Although the algorithm for the calculation of a correlation matrix between all terms in corpus is rather simple, there are some difficulties that reduce the effect of the correlation matrix for the purpose of, e.g., query expansion in information retrieval. An important question is, how to define the window in which co-occurrences between words are selected. Taking the whole document often yields a lot of accidental correlations, which are hard to filter out against seldom but significant correlations. Looking only at the sentence level misses a lot of counts for correlation which are often expressed beyond sentence boundaries by anaphoric constructions. In addition, not each co-occurance of two terms within a sentence, means that these words are necessarily semantically related.

To yield a precise sample of relations, we only count pairs of words, which are licensed by a set of predefined syntactic configurations. We used a simple regular grammar for German that yields a rather high precision, which we prefer in this situation over a high recall. The tradeoff of this approach is, that one yields even on corpus with over 40 million words, a lot of counts of pairs that are only observed two or three times, even if there exists a rather well known semantic relation between the terms.

We therefore used a two step algorithm, that first calculates the correlation between all pairs of words. In a second step, we use a clustering algorithm to determine correlation that are not directly visible in the sentences of the corpus but could be detected by comparing the context vectors of two terms. We call the first type of

correlation the syntacmatic correlations and the second type the paradigmatic correlations.

As an example for the importance of a paradigmatic relation, we observed in our intranet corpus a correlation between „text-mining“ and „speech-technology, because both terms share the same people and the same department as a common context, even if there is no sentence, where a direct relation between both terms can be found.

The result matrix of correlated terms is calculated as the weighted sum of both correlation values for each pair. While the first step can be calculated very efficient, since we use a threshold on the counts of the pairs, the second step has a quadratic complexity, since each term with all its contexts must be compared with all other terms and their contexts found in the corpora. After optimizing the database access and using thresholds on the term count, we were able to calculate the correlation matrix for the most frequent 50.000 words of the corpora in seven hours on a moderate Pentium computer with half a gigabyte RAM.

The matrix is visualized as a graph - or semantic network - with weighted edges according to the syntacmatic, paradicmatic or a overall correlation values. A user can type in a term, while the systems calculates all nodes with distance two and displays the resulting network using a Java applet GUI. The response time of the system is with less than a second for a query reasonable fast. Now the user can navigate the net by clicking on nodes to expand them, or directly search for a term, utilizing the search engine.

We add a so-called topic detection mechanism, which uses the semantic net as the basis for a query expansion mechanism. Thus, selecting a term for topic detection in the semantic network’s GUI, triggers a query expansion mechanism that selects a number of neighbor nodes in the net with the highest correlation with the terms and sends this set of pairs (which are connected with a NEAR operator) as a conjunction to the search engine. We use a kind of constraint relaxation mechanism, when the search engine does not find a result for the whole conjunction.

References

1. Lebeth, K. (2001): “Web-Data-Mining für Knowldeg-Management Projekte, Talk given at the 2. Data-Mining-Cup Conference in Chemnitz, URL: http://www.data-mining-cup.de/dmat/dmc2001_Lebeth.pdf
2. Martin Läuter, Uwe Quasthoff: Kollokationen und semantisches Clustering. In: Gippert, Jost; Olivier, Peter (edd.) (1999). Multilinguale Corpora. Codierung, Strukturierung, Analyse. Proc. 11. GLDV-Jahrestagung, Frankfurt/M. Prag: Enigma Corporation, 34-41.
3. Christopher D. Manning & Hinrich Schuetze (1999) Foundations of Statistical Natural Language Processing, MIT Press, Massachusetts, US

Collaborative Supply Net Management

Kurt Sundermeyer[1]

DaimlerChrysler AG, Research and Technology
Alt-Moabit 96 A
D-10559 Berlin, Germany
Kurt.Sundermeyer@daimlerchrysler.com

Abstract. With the advent of web technologies the supply logistics in industry received a boost in what is nowadays called eSupply, Supply Chain Management, or - even more appropriate – Supply Net Management (SNM). In this contribution research in this domain is motivated by sketching the complexity of SNM especially in automotive industry and by showing up possible pathways into a visionary future. Then essentially two current research tracks in Daimler-Chrysler R&T are addressed, namely (1) supply net configuration, simulation and assessment, and (2) distributed supply net planning. In both domains agent technology is used, partly together with other AI techniques.

1 Context and Complexity of Supply Net Management

These days car manufacturers are changing their businesses rapidly and radically. This is true both for the classical value chains beyond car manufacturing (e.g., distribution, after sales services etc.) and for the pre-assembly value chains, i.e., in the supply and procurement business [1].

In the center of an automotive OEM[2] are the assembly plants. These are provided on the first-tier level both by plants belonging to the OEM itself (e.g. engine plants) and by independent companies, the system suppliers. In principle this net goes up to the raw material. On the downstream side the assembly plants serve regional markets, retailers, and eventually the customer of the car, who is essentially pulling at the supply net.

To give an impression of the complexity of supply net management in Daimler-Chrysler automotive: more than 3000 suppliers (already 130 on the first-tier level), high degree of product complexity (about 10.000 parts/car), high degree of customization (esp. for supply parts), high volume production rate (up to 2.000 cars/ day and plant), long cycle times in some chains due to long lead times, various time and space constraints (w.r.t. capacity, stock size, part and component size).

---

[1] This contribution has more than one “father” in my research team. I especially thank Hartwig Baumgärtel and Jörg Wilke for their imagination and thorough work.

[2] Original Equipment Manufacturer, the driving element in the value chain

F. Baader, G. Brewka, and T. Eiter (Eds.): KI 2001, LNAI 2174, pp. 467–470, 2001.

Already very simple supply chains exhibit a surprising dynamics. So for instance in the MIT beer game [2], which consists of a linear chain with only four participants, simple changes in the demand of the consumer lead to rather unexpected stock and production policies at the nodes further down the chain. This demonstrates the bull-whip effect: forecasts and actual demand differ the more one goes down the chain.

Supply net management, has to cope with (1) A-synchronism: Effects related to decisions occur with long time-lags. (2) Non-locality: Effects of local decisions (often made with incomplete knowledge) occur at places far away from the point of decision making and are not visible at other decision points (3) Non-linearity: The net effect of the many parameters in the supply net on common objectives is not predictable. There are positive and negative feedback loops. Most quantities are bounded (capacities, batch sizes, safety stocks, buffers, ...)

2 From Isolated Enterprises to Agile Supply Communities

The current situation in the inter-related world of suppliers, manufacturers, sales, and customers is characterized by isolated planning islands. Transactions are treated with only minor responsiveness, without proper consideration of demand, and control is largely reactive. The information exchanged (e.g. by EDI) is semantically poor, and mainly limited to the transactional level.

One may envision a world [3] in which knowledge/content is exchanged (forecasts, orders, availability, shipments), where planning and optimization happens within the net, and where plans are adjusted and re-optimized in real-time. In this world a supply net is designed and operated for win-win situations of the OEM and its suppliers, change management and trouble shooting are utmost agile. There is order visibility in the sense of permanent evaluation and adaptation of capacity and delivery plans according to the current state of customer orders, production progress, inventory, and supply possibilities. This applies also downstream to the customer: She has the opportunity to use the delivery date as a configuration restriction of a vehicle ("configure-to-promise"), and she becomes informed by the dealer if problems arise in the production process, making transparent the decision alternatives.

This collaborative world can only be realized by (1) distributed SNM (2) integration of APS[3] systems across company borders, (3) inter-organizational supply net logistic control, (4) VPN[4] based planning data exchange. These requirements cry for AI, especially agent technology, since a supply scenario has all the characteristics for an agent system: It is distributed, with subsystems having a high degree of autonomy, and the interactions among the subsystems and with the environment are highly dynamic. However, agent technology alone is not the key to open the door to future SNM. As it is often the case for real-world applications, agent technology is to be combined with both other innovative techniques from AI (constraints, uncertainty reasoning, ontologies, data mining, ...) and with techniques from established logistics.

---

[3] Advanced Planning System, e.g. supply chain planning software like SAP/APO

[4] Virtual Private Network

3 Supply Net Configuration, Simulation, and Assessment

Our work in this domain aims at measuring the performance of collaborative sub-nets within the complete supply net. As of today, it is completely unclear, which kind of information exchange is appropriate for a good performance of the net. The essential logistic information resides in the BOM[5], and one of the decisive questions of a well-performing automotive supply net is "who does the BOM-explosion at which stage at which time to which granularity ?"

We try to find an answer to this question by simulation. The technical approach is to model supply net nodes and internal functions as software agents, each of them representing distributed planning domains. By this also proprietary and confidential data can be encapsulated. There are different agent types, each type characterized by specific patterns of behavior adjoined with specific sets of variables that are in the responsibility range of this very agent type. After being modeled, the dynamic simulation of the net receives e.g. demand functions and the output are supply net objectives. The simulation results then give hints how certain internal parameters of the net can be tuned to result in a leverage for supply net objectives.

The basic idea of this approach originates from the DASCh-System [4], which allows to configure and to simulate generic, linear supply chains. Together with ERIM/CEC[6] we extended this to a system which allows to simulate nets, and which respects certain requirements met in car industry. We were already successful in exhibiting qualitative process improvements [5]. By a further extension from the material flow to the logistics control level (which amounts to adding a further agent type) we are currently about to quantify the financial benefits for the partners involved. This is the foundation for inter-organizational supply net performance management.

4 Distributed Supply Net Planning

In principle it is imaginable that all relevant information about the status of a supply net is collected in a central data base, and that planning and optimization happens on this central unit. However, since independent enterprises are involved in SNM, it cannot be expected that all of them are willing to provide their data (exhibiting more or less explicitly also their policies) to a central instance. A company like Motorola for instance, being a supplier for Bosch, being a supplier for Mercedes-Benz, can hardly be motivated in displaying its planning data.

We follow an approach which explicitly respects local authority and responsibility. This was already successfully applied to capacity balancing over independent units in an assembly plant (so to say an OEM internal supply net [6]). In our approach we start from the consumer demands. These are propagated upstream in the net, and because of the network structure they can possibly be met in different ways. Depending on local

---

[5] Bill of Material

[6] ERIM/CEC is a research organization residing in Ann Arbor, Michigan.

capacities conflicts may arise. We model the units in the net by agents, which communicate upstream and downstream about their capacity constraints, possible constraint violations and their origin and, if it exists, find a solution for the demand requirements, or allow to track down hard capacity conflicts and trace their propagation.

5 Conclusion and Outlook: Collaborative Business

We believe that next generation eBiz is cBiz (collaborative Business). This is not only a new buzzword, but indicates a paradigm shift in logistics, in the sense that partners in the supply net will re-consider their roles and recognize that only by collaboration the win-win potential inherent in innovative supply net solutions can be realized.

Collaborative business has both technological and non-technological aspects. From the non-technological point, aside from the willingness to organizational changes, a feeling of trustful partnership must evolve. On the technology side we foresee three essential techniques. At first we believe that agent technology (for which collaboration is more or less inherent) is very appropriate to model, plan and run a supply network. And since at the very end everything boils down to bounded capacities at the different sites, constraint technology should be utilized. An important further strand are ontologies: The more semantically rich the context of SNM is described, the better is the quality of information exchanged by the supply net partners, and the more efficient is the collaboration.

References

1. Baumgärtel, H.: Zuliefernetzwerkorganisation im Zeichen von E-Business. In: Reiss, M. (Hrsg.): Netzwerkorganisation in der Unternehmenspraxis. Lemmens Verlags- und Mediengesellschaft, Bonn (2000) 37-58
2. Sterman, J.D.: Modeling Managerial Behavior: Misperceptions of Feedback in a Dynamic Decision Making Experiment. Management Science 35 (1989) 321-339
3. Christopher, M.: Logistics and Supply Chain Management. 2nd edn. Financial Time Prentice Hall, New Jersey (1999)
4. Parunak, V., Savit, R., Riolo, R.V.: Agent-Based Modeling vs. Equation-Based Modeling: A Case Study and Users' Guide: In: Proceedings of Multi-agent systems and Agent-based Simulation. Lecture Notes in Artificial Intelligence, Vol. 1534. Springer-Verlag, Berlin Heidelberg New York (1998) 10-25
5. Baumgärtel, H., Brückner, S., Parunak, V., Vanderbok, R., Wilke, J.: Agent Models of Supply Networks Dynamics. In: Billington, C., et al. (eds.): The Practice of Supply Chain Management. Int. Series on Operations Research and Management Science. Kluwer Academic Publ., New York (2001) forthcoming
6. Baumgärtel, H.: Verteiltes Lösen von Constraintproblemen in Multiagenten-Systemen zur optimierten Planung in einer Fließfertigung. Dissertationen zur Künstlichen Intelligenz (DISKI), Bd. 209. Infix, St. Augustin (1999)

Author Index