Lecture Notes in Computer Science 2401

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

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

Peter J. Stuckey (Ed.)

Logic Programming

18th International Conference, ICLP 2002
Copenhagen, Denmark, July 29 – August 1, 2002
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Peter J. Stuckey
University of Melbourne
Department of Computer Science and Software Engineering
221 Bouverie St., Carlton 3053, Australia
E-mail: pjs@s.mu.oz.au

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Logic programming : 18th international conference ; proceedings / ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002. Peter J. Stuckey (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2401)
ISBN 3-540-43930-7

CR Subject Classification (1998): I.2.3, D.1.6, D.3, F.3, F.4

ISSN 0302-9743
ISBN 3-540-43930-7 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

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

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein
Printed on acid-free paper SPIN 10873617 06/3142 5 4 3 2 1 0

Preface

These are the proceedings of the 18th International Conference on Logic Programming (ICLP 2002) held at the University of Copenhagen from July 29th to August 1st, 2002. For the first time the International Conference on Logic Programming was part of the Federated Logic Conference (FloC) which included the events:

- Conference on Automated Deduction (July 27-30)
- Conference on Computer-Aided Verification (July 27-31)
- Formal Methods Europe (July 22-24)
- IEEE Symposium on Logic in Computer Science (July 22-25)
- Conference on Rewriting Techniques and Applications (July 22-24)
- Automated Reasoning with Analytic Tableaux and Related Methods (July 30th - August 1st)

Furthermore seven satellite workshops were also associated with the conference and took place in the period July 27th to August 1st.

This volume contains the papers accepted for presentation at ICLP 2002. The conference received 82 submissions, in two categories: 21 applications papers and 61 technical papers. Of these, 29 papers were accepted for presentation, 7 applications papers and 22 technical papers. The program committee selected a best paper from each of the two categories. In addition 13 papers were accepted as short papers and presented as posters during the technical program.

The conference program also included two invited talks and four tutorials. We were privileged to have two distinguished invited speakers: Pierre Wolpe (University of Liege) talking about constraint representation by finite automata and Stefan Decker (Stanford University), talking about the possible roles of logic and logic programming in the Semantic Web. The four tutorials all represent areas of logic programming which are currently highly active. David D. Warren presented a tutorial on tabled logic programming, Thom Frühwirth and Slim Abdennadher presented constraint handling rules Mirek Truszczynski presented answer sets, and Abhik Roychoudhury and I.V. Ramakrishnan presented automated verification.

I would like to thank all the authors of the submitted papers, the Program Committee members, and the referees for their time and efforts spent in the reviewing process, the conference chair Henning Christiansen, and the FLoC organizers for the excellent organization of the conference.

May 2002 Peter J. Stuckey

Organization

ICLP 2002 was organized by the Association of Logic Programming

Conference Organization

Conference Chair:	Henning Christiansen (Roskilde University, Denmark)
Program Chair:	Peter J. Stuckey (University of Melbourne, Australia)
Workshop Chair:	Henning Christiansen (Roskilde University, Denmark)
FloC Conference Chair:	Neil D. Jones (Universty of Copenhagen, Denmark)
Publicity:	Martin Grohe (University of Edinburgh, UK)

Program Committee

José Alferes (Universidade Nova de Lisboa, Portugal)
Francisco Bueno (Universidad Polytecnica de Madrid, Spain)
Henning Christiansen (Roskilde University, Denmark)
Sandro Etalle (University of Twente, The Netherlands)
Francois Fages (INRIA, France)
Maurizio Gabbrielli (University of Bologna, Italy)
María García de la Banda (Monash University, Australia)
Michael Gelfond (Texas Tech University, USA)
Gopal Gupta (University of Texas at Dallas, USA)
Katsumi Inoue (Kobe University, Japan)
Joxan Jaffar (National University of Singapore, Singapore)
Gerda Janssens (Katholieke Universiteit Leuven, Belgium)
Bharat Jayaraman (State University of New York at Buffalo, USA)
Michael Leuschel (University of Southampton, UK)
Michael Maher (Loyola University, USA)
Dale Miller (Pennsylvania State University, USA)
Ulf Nilsson (Linköping University, Sweden)
Francesca Rossi (Università di Padova, Italy)
Konstantinos Sagonas (Uppsala University, Sweden)
Christian Schulte (Royal Institute of Technology, Sweden)
Harald Sondergaard (University of Melbourne, Australia)
Peter Stuckey (University of Melbourne, Australia)
Francesca Toni (Imperial College, UK)
Miroslaw Truszczyński (University of Kentucky, USA)
Pascal Van Hentenryck (Brown University, USA)
David S. Warren (State University of New York at Stony Brook, USA)

Referees

Areski Nait Abdallah
Salvador Abreu
Gianluca Amato
James Bailey
Marcello Balduccini
Chitta Baral
David Billington
Roland Bol
Lucas Bordeaux
Maurice Bruynooghe
Daniel Cabeza
Manuel Carro
Carlo Combi
Emmanuel Coquery
Jesús Correas
Carlos Damasio
Frank de Boer
Danny De Schreye
Alex Dekhtyar
Bart Demoen
Jürgen Dix
Mireille Ducassé
Ines Dutra
Artur Garcez
Ana García-Serrano
José Manuel Gómez
Roberta Gori
Michel Grabisch
Stefan Gruner
Sergio Guadarrama
Hai-Feng Guo
Martin Henz
Koji Iwanuma
Nico Jacobs
Antonis Kakas
Bob Kowalski
Kung-Kiu Lau
João Leite
Lengning Liu
Pedro López-García
Jan Małuszyński
José Manuel Gómez
Massimo Marchiori
Victor W. Marek
Kim Marriott
Nancy Mazur
Chiara Meo
Laurent Michel
Tim Miller
J. J. Moreno Navarro
S. Muñoz Hernández
Ilkka Niemelä
Nicolay Pelov
Luís Moniz Pereira
Luis Pias de Castro
Paweł Pietrzak
Iman Hafiz Poernomo
Enrico Pontelli
Steven Prestwich
Germán Puebla
Jan Ramon
Nico Roos
Abhik RoyChoudhury
Salvatore Ruggieri
Fariba Sadri
Chiaki Sakama
Vítor Santos Costa
Taisuke Sato
Peter Schachte
Andrea Schaerf
Alexander Serebrenik
Kish Shen
Fernando Silva
Zoltan Somogyi
Fausto Spoto
Kostas Stathis
Gheorghe Stefanescu
Martin Sulzmann
P.S. Thiagarajan
Paolo Torroni
Son Cao Tran
Claudio Vaucheret
Eric de la Clergerie
Eelco Visser
Razvan Voicu
Richard Watson
Herbert Wiklicky
Cees Witteveen
Roland Yap
Jia-Huai You

Prizes

Best Paper (Applications Program)
An abductive approach for analysing event-based requirements specifications,
Alessandra Russo, Rob Miller, Bashar Nuseibeh, and Jeff Kramer.
Best Paper (Technical Program)
Trailing analysis for HAL,
Tom Schrijvers, María García de la Banda, and Bart Demoen.

Sponsors

IF Computer GmbH
Association of Logic Programming
Federated Logic Conference (FLoC02)
University of Copenhagen

Table of Contents

Invited Speakers

Representing Arithmetic Constraints with Finite Automata: An Overview ... 1
Bernard Boigelot and Pierre Wolper

Logic Databases on the Semantic Web: Challenges and Opportunities 20
Stefan Decker

Conference Papers

An Abductive Approach
for Analysing Event-Based Requirements Specifications 22
Alessandra Russo, Rob Miller, Bashar Nuseibeh, and Jeff Kramer

Trailing Analysis for HAL 38
Tom Schrijvers, Maria García de la Banda, and Bart Demoen

Access Control for Deductive Databases by Logic Programming 54
Steve Barker

Reasoning about Actions with CHRs and Finite Domain Constraints 70
Michael Thielscher

Using Hybrid Concurrent Constraint Programming
to Model Dynamic Biological Systems 85
Alexander Bockmayr and Arnaud Courtois

Efficient Real-Time Model Checking
Using Tabled Logic Programming and Constraints 100
G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan

Constraint-Based Infinite Model Checking
and Tabulation for Stratified CLP 115
Witold Charatonik, Supratik Mukhopadhyay, and Andreas Podelski

A Model Theoretic Semantics
for Multi-level Secure Deductive Databases 130
Hasan M. Jamil and Gillian Dobbie

Propagation Completeness of Reactive Constraints 148
Michael J. Maher

On Enabling the WAM with Region Support 163
Henning Makholm and Konstantinos Sagonas

A Different Look at Garbage Collection for the WAM 179
Bart Demoen

Copying Garbage Collection for the WAM: To Mark or Not to Mark? 194
Bart Demoen, Phuong-Lan Nguyen, and Ruben Vandeginste

Logical Algorithms 209
Harald Ganzinger and David McAllester

Logical Loops 224
Joachim Schimpf

Learning in Logic with RichProlog 239
Eric Martin, Phuong Nguyen, Arun Sharma, and Frank Stephan

Towards a Declarative Query and Transformation Language
for XML and Semistructured Data: Simulation Unification 255
François Bry and Sebastian Schaffert

A Proof-Theoretic Foundation
for Tabled Higher-Order Logic Programming 271
Brigitte Pientka

Proving the Equivalence of CLP Programs 287
Sorin Craciunescu

A Purely Logical Account of Sequentiality in Proof Search 302
Paola Bruscoli

Disjunctive Explanations 317
Katsumi Inoue and Chiaki Sakama

Reasoning with Infinite Stable Models II: Disjunctive Programs 333
Piero A. Bonatti

Computing Stable Models: Worst-Case Performance Estimates 347
Zbigniew Lonc and Mirosław Truszczyński

Towards Local Search for Answer Sets 363
Yannis Dimopoulos and Andreas Sideris

A Rewriting Method for Well-Founded Semantics with Explicit Negation ..378
Pedro Cabalar

Embedding Defeasible Logic into Logic Programs 393
Grigoris Antoniou and Michael J. Maher

A Polynomial Translation of Logic Programs with Nested Expressions
into Disjunctive Logic Programs: Preliminary Report 405
*David Pearce, Vladimir Sarsakov, Torsten Schaub, Hans Tompits,
and Stefan Woltran*

Using Logic Programming to Detect Activities in Pervasive Healthcare 421
Henrik Bærbak Christensen

Logic Programming for Software Engineering: A Second Chance 437
Kung-Kiu Lau and Michel Vanden Bossche

A Logic-Based System for Application Integration 452
Tamás Benkő, Péter Krauth, and Péter Szeredi

Conference Papers

The Limits of Horn Logic Programs 467
Shilong Ma, Yuefei Sui, and Ke Xu

Multi-adjoint Logic Programming: A Neural Net Approach 468
Jesús Medina, Enrique Mérida-Casermeiro, and Manuel Ojeda-Aciego

Fuzzy Prolog: A Simple General Implementation Using CLP($\mathcal{R}$) 469
Claudio Vaucheret, Sergio Guadarrama, and Susana Muñoz

Automated Analysis of CLP(FD) Program Execution Traces 470
Mireille Ducassé and Ludovic Langevine

Schema-Based Transformations of Logic Programs in λProlog 472
Petr Olmer and Petr Štěpánek

Non-uniform Hypothesis in Deductive Databases with Uncertainty 473
Yann Loyer and Umberto Straccia

Probabilistic Finite Domains: A Brief Overview 475
Nicos Angelopoulos

Modelling Multi-agent Reactive Systems 476
Prahladavaradan Sampath

Integrating Planning, Action Execution, Knowledge Updates and Plan Modifications via Logic Programming 477
Hisashi Hayashi, Kenta Cho, and Akihiko Ohsuga

A Logic Program Characterization of Domain Reduction Approximations in Finite Domain CSPs 478
Gérard Ferrand and Arnaud Lallouet

TCLP: Overloading, Subtyping and Parametric Polymorphism Made Practical for CLP 480
Emmanuel Coquery and François Fages

Logical Grammars Based on Constraint Handling Rules 481
Henning Christiansen

Debugging in A-Prolog: A Logical Approach 482
Mauricio Osorio, Juan Antonio Navarro, and José Arrazola

Author Index 485

Representing Arithmetic Constraints with Finite Automata: An Overview*

Bernard Boigelot and Pierre Wolper

Institut Montefiore, B28, Université de Liège
4000 Liège, Belgium
{boigelot,pw}@montefiore.ulg.ac.be
http://www.montefiore.ulg.ac.be/~{boigelot,pw}

Abstract. Linear numerical constraints and their first-order theory, whether defined over the reals or the integers, are basic tools that appear in many areas of Computer Science. This paper overviews a set of techniques based on finite automata that lead to decision procedures and other useful algorithms, as well as to a normal form, for the first-order linear theory of the integers, of the reals, and of the integers and reals combined. This approach has led to an implemented tool, which has the so far unique capability of handling the linear first-order theory of the integers and reals combined.

1 Introduction

Linear numerical constraints, i.e. constraints involving only addition or multiplication by constants, are a basic tool used in many areas of Computer Science and other disciplines. There is thus an abundance of algorithms and tools dealing with linear constraints, which mostly are geared to efficiently solving consistency and optimization problems. The power of linear constraints can be significantly enhanced if they are incorporated in a first-order theory allowing Boolean operations and quantification. But, this comes at the price of higher complexity, and tools handling the full first-order theory are less common, especially when the constraints are defined over the integers, the latter case corresponding to Presburger arithmetic, a decidable but two exponential-space complete theory. This limited number of tools was, until the approach described in this paper, even a complete absence when moving to the first-order theory of linear constraints over the reals and integers *combined*, i.e. involving both variables ranging over the reals and variables ranging over the integers.

The work overviewed here was motivated by problems related to the symbolic exploration of infinite state spaces [WB98], for which handling nonconvex and periodic constraints over the integers was essential. A general Presburger tool (for instance [Pug92]) was in principle sufficient for the task. However, the need

* This work was partially funded by a grant of the "Communauté française de Belgique - Direction de la recherche scientifique - Actions de recherche concertées" and by the European IST-FET project ADVANCE (IST-1999-29082).

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 1–20, 2002.
© Springer-Verlag Berlin Heidelberg 2002

to very frequently check implication of formulas, as well as the success of Binary Decision Diagrams (BDDs) [Bry86] for similarly dealing with Boolean formulas in the symbolic exploration of finite state spaces [BCM+90], prompted us to search for a related representation of arithmetic formulas. The long known fact that by encoding integers as binary (or in general r-ary) strings, Presburger arithmetic can be handled by finite automata [Büc60], naturally pointed to finite automata as a tool for dealing with arithmetic, and to minimized finite automata as a normal form akin to BDDs. From a theoretical point of view this was all pretty obvious, what still needed to be done was to turn this idea into a working technology. The first step towards this was a careful choice of the coding of numbers by strings, for instance using r's complement for negative numbers and sequentializing the bits of vectors (see Section 3). Second, the development of specific algorithms, e.g. for generating automata directly from equations and inequations [BC96, WB00] was very helpful. Finally, an efficient package for dealing with finite automata, the LASH tool [LAS] was developed. Besides providing a Presburger tool, LASH also included some specific components to deal with state-space exploration, for instance an algorithm for computing (when possible) the effect of iterating a linear transformation [Boi98].

If automata on finite words can handle integer arithmetic, it follows almost immediately that automata on infinite words [Tho90] can handle real arithmetic. This has also been long known, but turning this attractive idea into a technology was substantially more difficult since manipulating automata on infinite words requires less efficient and harder to implement algorithms than for automata on finite words. However, it turns out that handling real linear arithmetic with automata does not require the full power of infinite-word automata, but can be done with a very restrictive class, namely deterministic weak automata [Sta83, MSS86, MS97]. This was shown using topological arguments in [BJW01] with two important consequences: algorithms very similar to those used for finite-word automata can be used for manipulating this class, and it admits a easily computable reduced normal form [Löd01]. Now, since it is very easy to express that a number is an integer with an automaton (its fractional part is 0), the automata-theoretic approach to handling real arithmetic can also cope with the theory in which both real and integer variables are allowed. This makes it for example possible to represent an infinite periodic set of dense intervals, something that is beyond the linear first-order theory of the reals alone. Potential applications include, for instance, the analysis of some classes of hybrid systems [BBR97].

The goal of this paper is to present an overview of the theory and pragmatics of handling integer and real arithmetic with automata. Since the case of pure integer arithmetic is simply the restriction of the real case to finite words, only the more general latter case is presented, simplifications that occur in the pure integer case being mentioned. After a section recalling the necessary definitions about automata on infinite words, the encoding scheme by which a set of real vectors can be represented by a finite automaton accepting a set of infinite words is presented. We then give the algorithms for directly constructing

automata from linear equations and inequations. Next, the automata-theoretic operations corresponding to the first-order logical constructs are reviewed and the corresponding algorithms are described. Finally, a series of experimental results obtained with the LASH tool are presented and some conclusions are given.

2 Logical and Automata-Theoretic Background

In this section we recall some logical and automata-theoretic concepts that are used in the paper.

2.1 Theories of the Integers and Reals

The main theory we will consider in this paper is the first-order theory of the structure $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, where $+$ represents the predicate $x + y = z$. Since any linear equality or order constraint can be encoded into this theory, we refer to it as additive or linear arithmetic over the reals and integers. We will often refer to its restriction to integer variables as Presburger arithmetic, though the theory originally defined by Presburger was defined over the natural numbers.

2.2 Automata on Infinite Words

An infinite word (or ω-word) w over an alphabet Σ is a mapping $w : \mathbb{N} \to \Sigma$ from the natural numbers to Σ. A Büchi automaton on infinite words is a five-tuple $A = (Q, \Sigma, \delta, Q_0, F)$, where

- Q is a finite set of states;
- Σ is the input alphabet;
- δ is the transition function and is of the form $\delta : Q \times \Sigma \to 2^Q$ if the automaton is nondeterministic and of the form $\delta : Q \times \Sigma \to Q$ if the automaton is deterministic;
- $Q_0 \subseteq Q$ is a set of initial states (a singleton for deterministic automata);
- F is a set of accepting states.

A run π of a Büchi automaton $A = (Q, \Sigma, \delta, q_0, F)$ on an ω-word w is a mapping $\pi : \mathbb{N} \to Q$ that satisfies the following conditions:

- $\pi(0) \in Q_0$, i.e. the run starts in an initial state;
- For all $i \geq 0$, $\pi(i+1) \in \delta(\pi(i), w(i))$ (nondeterministic automata) or $\pi(i+1) = \delta(\pi(i), w(i))$ (deterministic automata), i.e. the run respects the transition function.

Let $inf(\pi)$ be the set of states that occur infinitely often in a run π. A run π is said to be accepting if $inf(\pi) \cap F \not= \emptyset$. An ω-word w is accepted by a Büchi automaton if that automaton has some accepting run on w. The language $L_\omega(A)$ of infinite words defined by a Büchi automaton A is the set of ω-words it accepts.

A co-Büchi automaton is defined exactly as a Büchi automaton except that its accepting runs are those for which $inf(\pi) \cap F = \emptyset$.

We will also use the notion of *weak* automata [MSS86]. For a Büchi automaton $A = (Q, \Sigma, \delta, Q_0, F)$ to be *weak*, there has to be a partition of its state set Q into disjoint subsets $Q_1, \ldots, Q_m$ such that

- for each of the Q_i either $Q_i \subseteq F$ or $Q_i \cap F = \emptyset$; and
- there is a partial order $\leq$ on the sets $Q_1, \ldots, Q_m$ such that for every $q \in Q_i$ and $q' \in Q_j$ for which, for some $a \in \Sigma$, $q' \in \delta(q, a)$ ($q' = \delta(q, a)$ in the deterministic case), $Q_j \leq Q_i$.

For more details, a survey of automata on infinite words can be found in [Tho90].

3 Representing Sets of Integers and Reals with Finite Automata

In order to use a finite automaton for recognizing numbers, one needs to establish a mapping between these and words. Our encoding scheme corresponds to the usual notation for reals and relies on an arbitrary integer base $r > 1$. We encode a number x in base r, most significant digit first, by words of the form $w_I \star w_F$, where w_I encodes the integer part x_I of x as a finite word over $\{0, \ldots, r-1\}$, the special symbol "$\star$" is a separator, and w_F encodes the fractional part x_F of x as an infinite word over $\{0, \ldots, r-1\}$. Negative numbers are represented by their r's complement. In this notation, a number $d_k d_{k-1} \ldots d_1 d_0 \star d_{-1} d_{-2} \ldots$ written in base r and of integer length $k+1$ is positive if it starts with 0 and negative if it starts with $r-1$, in which case its value is $-r^{k+1} + \sum_{-\infty < i \leq k} d_i r^i$. Since we are dealing with encodings of variable length, we need to make sure that the number of digits used for the integer part is sufficient for the leading digit always to be 0 for positive numbers and $r-1$ for negative numbers. This is done by requiring that the number of digits $k+1$ of the integer part of a number x be such that $-r^k \leq x < r^k$ ($-r^k \leq x \leq r^k$ if the encoding of the fractional part of x is an infinite sequence of digits $r-1$). Note that repeating the leading digit has no impact on the value of the number since it is 0 for positive numbers and since, for negative numbers, we have that $-r^k + (r-1)r^{k-1} = -r^{k-1}$.

According to this scheme, each number has an infinite number of encodings, given that the integer-part length can be increased unboundedly. In addition, the rational numbers whose denominator has only prime factors that are also factors of r have two distinct encodings with the same integer-part length. For example, in base 10, the number $11/2$ has the encodings $005 \star 5(0)^\omega$ and $005 \star 4(9)^\omega$, "ω" denoting infinite repetition.

To encode a vector of real numbers, we represent each of its components by words of identical integer-part length. This length can be chosen arbitrarily, provided that it is sufficient for encoding the vector component with the highest magnitude. An encoding of a vector $\boldsymbol{x} \in \mathbb{R}^n$ can indifferently be viewed either as a n-tuple of words of identical integer-part length over the alphabet $\{0, \ldots, r-1, \star\}$, or as a single word w over the alphabet $\{0, \ldots, r-1\}^n \cup \{\star\}$.

Example 1. In base 2, the vector $(-2, 12.3)$ can be encoded as

$$(11110 \star 0^\omega, 01100 \star 01[1001]^\omega)$$

or as the word

$$(1,0)(1,1)(1,1)(1,0)(0,0) \star (0,0)(0,1)[(0,1)(0,0)(0,0)(0,1)]^\omega.$$

Using an alphabet of size r^n is clearly going to be problematic as soon as n starts to grow. The solution is to read the digits of the various components of the vector serially, in a round robin way, thus reducing the alphabet size to the perfectly manageable r. We will refer to this scheme as the *serial encoding* as opposed to the *simultaneous encoding* in which the alphabet consists of tuples of digits.

Example 2. Using the serial encoding, the vector $(-2, 12.3)$ can be encoded in base 2 as

$$1011111000 \star 0001[01000001]^\omega.$$

Real vectors being encoded by infinite words, a set of vectors can be represented by an infinite-word automaton accepting the corresponding encodings. However, since a real vector has an infinite number of possible encodings, we have to choose which of these the automata will recognize. A natural choice is to accept all encodings. This leads to the following definition.

Definition 1. *Let $n > 0$ and $r > 1$ be integers. A base-r n-dimension serial* Real Vector Automaton (RVA) *is a Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ automaton over the alphabet $\Sigma = \{0, \ldots, r-1\} \cup \{\star\}$, such that*

- *Every word accepted by $\mathcal{A}$ is a serial encoding in base r of a vector in $\mathbb{R}^n$, and*
- *For every vector $\boldsymbol{x} \in \mathbb{R}^n$, $\mathcal{A}$ accepts either all the encodings of $\boldsymbol{x}$ in base r, or none of them.*

An RVA is said to *represent* the set of vectors encoded by the words that belong to its accepted language.

From a theoretical point of view, there is no difference between the serial and simultaneous encodings, and it is easy to move from one to the other. From an implementation point of view, using the serial encoding is clearly preferable. Notice also, that in the context of minimized deterministic automata, the serial encoding is essentially equivalent to using the simultaneous scheme, while representing the transitions from a state to another by a BDD (an r-ary Decision Diagram for bases r other than 2). The expressive power of RVAs has been studied in [BRW98] and corresponds exactly to linear arithmetic over the reals and integers, extended with a special base-dependent predicate that can check the value of the digit appearing in a given position. If this special predicate is not used, RVAs can always be constructed to be weak automata [BJW01], which we will always assume to be the case in what follows.

4 Constructing Automata from Linear Relations

To construct automata corresponding to the formulas of linear arithmetic, we start with automata corresponding to linear equalities and inequalities as basic building blocks. It would of course also be possible to start just with automata for addition and order comparison, but there are simple and easy to implement constructions [BC96, BRW98, WB00] that directly produce close to optimal deterministic automata for linear relations, which explains our choice. This section describes these constructions in the case of real variables and the serial encoding of vectors. If both integer and real variables are involved in a relation, integerhood can be imposed by forcing the fractional part of the integer variables to be either 0^ω or $(r-1)^\omega$. This can be done by a simple adaptation of the constructions below, or by imposing integerhood as a separate constraint for which a specific automaton is constructed.

4.1 Linear Equations

The problem addressed consists of constructing an RVA that represents the set S of all the solutions $\boldsymbol{x} \in \mathbb{R}^n$ of an equation of the form $\boldsymbol{a}.\boldsymbol{x} = b$, given $n \geq 0$, $\boldsymbol{a} \in \mathbb{Z}^n$ and $b \in \mathbb{Z}$.

A Decomposition of the Problem The basic idea is to build the automaton corresponding to a linear equation in two parts : one that accepts the integer part of solutions of the equation and one that accepts the part of the solution that belongs to $[0,1]^n$. For convenience, in what follows the n-dimension real vector represented by a word w interpreted as a serial encoding in base r is denoted $[w]_r^n$.

More precisely, let $\boldsymbol{x} \in S$, and let $w_I \star w_F$ be a serial encoding of $\boldsymbol{x}$ in a base $r > 1$, with $w_I \in \Sigma^+$, $w_F \in \Sigma^\omega$, and $\Sigma = \{0, \ldots, r-1\}$. The vectors $\boldsymbol{x}_I$ and $\boldsymbol{x}_F$, respectively encoded by the words $w_I \star 0^\omega$ and $0^n \star w_F$, are such that $\boldsymbol{x}_I \in \mathbb{Z}^n$, $\boldsymbol{x}_F \in [0,1]^n$, and $\boldsymbol{x} = \boldsymbol{x}_I + \boldsymbol{x}_F$. Since $\boldsymbol{a}.\boldsymbol{x} = b$, we have $\boldsymbol{a}.\boldsymbol{x}_I + \boldsymbol{a}.\boldsymbol{x}_F = b$. Moreover, writing $\boldsymbol{a}$ as $(a_1, \ldots, a_n)$, we have $\alpha \leq \boldsymbol{a}.\boldsymbol{x}_F \leq \alpha'$, where $\alpha = \sum_{a_i<0} a_i$ and $\alpha' = \sum_{a_i>0} a_i$, which implies $b - \alpha' \leq \boldsymbol{a}.\boldsymbol{x}_I \leq b - \alpha$. Another immediate property of interest is that $\boldsymbol{a}.\boldsymbol{x}_I$ is divisible by $\gcd(a_1, \ldots, a_n)$.

From those results, we obtain that the language L of the encodings of all the elements of S satisfies

$$L = \bigcup_{\varphi(\beta)} \{w_I \in \Sigma^+ \mid \boldsymbol{a}.[w_I \star 0^\omega]_r^n = \beta\} \cdot \{\star\} \cdot \{w_F \in \Sigma^\omega \mid \boldsymbol{a}.[0^n \star w_F]_r^n = b - \beta\},$$

where "$\cdot$" denotes concatenation and $\varphi(\beta)$ stands for $b - \alpha' \leq \beta \leq b - \alpha \wedge (\exists m \in \mathbb{Z})(\beta = m \gcd(a_1, \ldots, a_n))$.

This decomposition of L reduces the computation of an RVA representing S to the following problems:

- building an automaton on finite words accepting all the words $w_I \in \Sigma^+$ such that $[w_I \star \mathbf{0}^\omega]_r^n$ is a solution of a given linear equation;

– building a Büchi automaton accepting all the words $w_F \in \Sigma^\omega$ such that $[0^n \star w_F]_r^n$ is a solution of a given linear equation.

These problems are addressed in the two following sections.

Recognizing Integer Solutions Our goal is, given an equation $\boldsymbol{a}.\boldsymbol{x} = b$ where $\boldsymbol{a} = (a_1, \ldots, a_n) \in \mathbb{Z}^n$ and $b \in \mathbb{Z}$, to construct a finite automaton $\mathcal{A}_{\boldsymbol{a},b}$ that accepts all the finite words serially encoding in a given base r the integer solutions of that equation.

The construction proceeds as follows. Except for the unique initial state s_0 and the states reached from there while reading the first n digits of the vector encoding (the sign digits), the states s of $\mathcal{A}_{\boldsymbol{a},b}$ are in one-to-one correspondence with a pair (γ, i), where γ is an integer and $0 \leq i \leq n-1$ denotes a position in the serial reading of the vector digits. A state s of the form $(\gamma, 0)$, corresponding to the situation in which the number of digits read is the same for each vector component, has the property that the vectors $\boldsymbol{x} \in \mathbb{Z}^n$ accepted by the paths leading from s_0 to s are exactly the solutions of the equation $\boldsymbol{a}.\boldsymbol{x} = \gamma$. The only accepting state s_F of $\mathcal{A}_{\boldsymbol{a},b}$ is $(b, 0)$.

The next step is to define the transitions of $\mathcal{A}_{\boldsymbol{a},b}$. Consider first moving from a state s of the form $(\gamma, 0)$. The next n digits $d_1, \ldots, d_n$ that will be read lengthen the encoding of each component of the vector by 1 and hence, if $\boldsymbol{x}$ is the value of the vector read before inputing the digits $d_1, \ldots, d_n$, then its value $\boldsymbol{x'}$ after reading these digits is $\boldsymbol{x'} = r\boldsymbol{x} + (d_1, \ldots, d_n)$. If only the first $i < n$ of the digits $d_1, \ldots, d_n$ have been read, the value of the vector will be taken to be $\boldsymbol{x'} = r\boldsymbol{x} + (d_1, \ldots, d_i, 0, \ldots, 0)$. Therefore, for a state s of the form (γ, i), an outgoing transition labeled d must lead to a state $s' = (\gamma', (i+1) \bmod n)$ such that $\gamma' = r\gamma + a_1 d$ if $i = 0$ and $\gamma' = \gamma + a_{i+1} d$ if $i > 0$.

For transitions from the single initial state s_0, one has to take into account the fact the the first n digits read are the sign bits of the vector components. Thus a digit $r-1$ should be interpreted as -1, no other digit except 0 being allowed. The states other than s_0 reached during the reading of the sign digits will be characterized, as all other states, by a value and a position in the serial reading of digits, but this position will be represented as a negative number $-(n-1) \leq i \leq -1$ in order to distinguish these states from the states reached after the sign digits have been read. Thus a transition labeled $d \in \{0, r-1\}$ from s_0 leads to the state $s' = (-a_1, -1)$ if $d = r-1$, and $s' = (0, -1)$ if $d = 0$. Similarly, a transition labeled $d \in \{0, r-1\}$, from a state $s = (\gamma, -i)$ leads to the state $s' = (\gamma - a_{i+1}, -((i+1) \bmod n))$ if $d = r-1$, and $s' = (\gamma, -((i+1) \bmod n))$ if $d = 0$.

Notice that the transition relation we have defined is deterministic and that only a finite number of states are needed. Indeed, from a state $s = (\gamma, 0)$ such that $|\gamma| > (r-1)\sum_{i=1}^n |a_i|$, one can only reach states s' such that $|\gamma'| > |\gamma|$, hence all states $s = (\gamma, 0)$ such that $|\gamma| > (r-1)\sum_{i=1}^n |a_i|$ and $|\gamma| > |b|$, as well as their successors, can be pruned. If all unnecessary states (those from which the accepting state cannot be reached) are pruned, the automaton obtained is, within a small exception, a minimal deterministic automaton. Indeed, if it was

possible to merge two states $s = (\gamma, i)$ and $s' = (\gamma', i)$ with $\gamma' \not= \gamma$, then different right-hand sides for the equation $\boldsymbol{a}.\boldsymbol{x} = \beta$ would yield identical solutions, a contradiction. However, absolute minimality is not guaranteed since it could still be possible to merge a state (γ, i) with the state $(\gamma, -i)$ without changing the accepted language.

In practice, to avoid the pruning of unnecessary states, it is convenient to compute the automaton $\mathcal{A}_{\boldsymbol{a},b}$ backwards, starting from the accepting state $s_F = (b, 0)$. Computing the backwards transitions is quite straightforward. Indeed, an incoming d-labeled transitions to a state $s = (\gamma, i)$ with $1 < i \leq n-1$ has to originate from the state $s' = (\gamma - a_i d, i-1)$. If $i = 0$, the origin state is $s' = (\gamma - a_n d, n-1)$, and, if $i = 1$ it is $s' = ((\gamma - a_1 d)/r, 0)$. If $(\gamma - a_1 d)/r$ is not an integer or is not divisible by $\gcd(a_1, \ldots, a_n)$, the state s' should not be created, and the states only reachable from it should be pruned. Finally, from a state s of the form $(\gamma, 0)$, one should also consider the possibility that the digits read to reach it are the sign digits. This is only possible if $\exists \sigma \subseteq \{a_1, \ldots, a_n\}\ \gamma = -\sum_{a_i \in \sigma} a_i$, in which case one should also move backwards from the state $(\gamma, 0)$ to $(\gamma + a_n, -(n-1))$, or $(\gamma, -(n-1))$, and so on for all possibilities that finally go back to the unique initial state s_0.

If one wishes to construct k automata $\mathcal{A}_{\boldsymbol{a},b_1}, \mathcal{A}_{\boldsymbol{a},b_2}, \ldots, \mathcal{A}_{\boldsymbol{a},b_k}$ with $b_1, \ldots, b_k \in \mathbb{Z}$ (for instance, as an application of the method presented in Section 4.1, in which the b_i are all the integers satisfying φ), then a technique more efficient than repeating the construction k times consists of starting from the set $\{(b_1, 0), \ldots, (b_k, 0)\}$, rather than from a set containing a single state. The states and transitions computed during the construction will then be shared between the different $\mathcal{A}_{\boldsymbol{a},b_i}$, and each $(b_i, 0)$ will be the only accepting state of the corresponding $\mathcal{A}_{\boldsymbol{a},b_i}$.

As is shown in [BRW98], the number of states of the automaton constructed for an equation $\boldsymbol{a}.\boldsymbol{x} = b$ is logarithmic in the value of b and linear in the absolute value of the elements of $\boldsymbol{a}$. Finally, note that for any equation and base there is a unique minimal deterministic automaton accepting the solutions of the equation. It can always be obtained by applying standard minimization techniques [Hop71] to the automaton obtained from the construction described above or, as a matter of fact, to any automaton accepting the same language.

Recognizing Fractional Solutions We now address the computation of a Büchi automaton $\mathcal{A}'_{\boldsymbol{a},b}$ that accepts all the infinite words $w \in \Sigma^\omega$ such that $0^n \star w$ encodes a solution $\boldsymbol{x} \in [0, 1]^n$ of the equation $\boldsymbol{a}.\boldsymbol{x} = b$.

The construction is similar to the one of the previous section, except that we are now dealing with the expansion of fractional numbers. The states s of $\mathcal{A}'_{\boldsymbol{a},b}$ are in a one-to-one correspondence with a pair (γ, i), where γ is an integer and $0 \leq i \leq n-1$ denotes a position in the serial reading of the vector digits. A state s of the form $(\gamma, 0)$, corresponding to the situation in which the number of digits read is the same for each vector component, has the property that the vectors $\boldsymbol{x} \in [0, 1]^n$ accepted by the infinite paths starting from s are exactly the

solutions of the equation $\boldsymbol{a}.\boldsymbol{x} = \gamma$. The set of initial states contains only the pair $(b, 0)$. All the states are accepting.

The transitions of $\mathcal{A}'_{\boldsymbol{a},b}$ are defined as follows. Consider first moving from a state s of the form $(\gamma, 0)$. The next n digits $d_1, \ldots, d_n$ that will be read lengthen the encoding of each component of the vector by 1. This amounts to prefixing the digits $d_1, \ldots, d_n$ to the word that will be read from next state s' of the form $(\gamma', 0)$ that will be reached. The value $\boldsymbol{x}$ of the word read from s is thus related to the value $\boldsymbol{x}'$ of the word read from s' by $\boldsymbol{x} = (1/r)(\boldsymbol{x}' + (d_1, \ldots, d_n))$, which can be rewritten as $\boldsymbol{x}' = r\boldsymbol{x} - (d_1, \ldots, d_n)$. If only the first $i < n$ of the digits $d_1, \ldots, d_n$ have been read, the value of the vector will be taken to be $\boldsymbol{x}' = r\boldsymbol{x} + (d_1, \ldots, d_i, 0, \ldots, 0)$. Therefore, for a state s of the form (γ, i), an outgoing transition labeled d must lead to a state $s' = (\gamma', (i + 1) \bmod n)$ such that $\gamma' = r\gamma - a_1 d$ if $i = 0$ and $\gamma' = \gamma - a_{i+1} d$ if $i > 0$.

Note that for states of the form $(\gamma, 0)$, γ must belong to the interval $[\alpha, \alpha']$, where α and α' are as defined in Section 4.1, otherwise there would be no solution in $[0, 1]^n$ to $\boldsymbol{a}.\boldsymbol{x} = \gamma$. Only a finite number of states are thus necessary.

The automaton $\mathcal{A}'_{\boldsymbol{a},b}$ can be constructed by starting from the state $s = (b, 0)$, and then repeatedly computing the outgoing transitions from the current states until stabilization occurs. Like in Section 4.1, the construction of k automata $\mathcal{A}'_{\boldsymbol{a},b_1}, \mathcal{A}'_{\boldsymbol{a},b_2}, \ldots, \mathcal{A}'_{\boldsymbol{a},b_k}$, with $b_1, \ldots, b_k \in \mathbb{Z}$ (for instance, as an application of the method presented in Section 4.1) can simply be done by starting from the set $\{(b_1, 0), \ldots, (b_k, 0)\}$, rather than from a set containing a single state. The computation terminates, since for every state of the form $s = (\gamma, 0)$, the integer γ belongs to the bounded interval $[\alpha, \alpha']$. Once dead-end states are pruned, the automaton obtained is deterministic. Notice also that it is weak since all its states are accepting and can be minimized by the procedure sketched in Section 5.4. Interestingly, because we are dealing with automata on infinite words, states (γ, i) and (γ, j) with $i \not= j$ can occasionally be merged, which precludes the construction above from always producing a minimal automaton.

4.2 Linear Inequations

The method presented for equations can be easily adapted to linear inequations. The problem consists of computing an RVA representing the set of all the solutions $\boldsymbol{x} \in \mathbb{R}^n$ of an inequation of the form $\boldsymbol{a}.\boldsymbol{x} \leq b$, given $n \geq 0$, $\boldsymbol{a} \in \mathbb{Z}^n$ and $b \in \mathbb{Z}$.

The decomposition of the problem into the computation of representations of the sets of integer solutions and of solutions in $[0, 1]^n$ of linear inequations is identical to the one proposed for equations in Section 4.1.

Given an inequation of the form $\boldsymbol{a}.\boldsymbol{x} \leq b$, where $\boldsymbol{a} \in \mathbb{Z}^n$ and $b \in \mathbb{Z}$, the definition of an automaton $\mathcal{A}_{\boldsymbol{a},b}$ that accepts all the finite words $w \in \Sigma^*$ such that $w \star 0^\omega$ encodes an integer solution of $\boldsymbol{a}.\boldsymbol{x} \leq b$ is very similar to the one given for equations in Section 4.1. Indeed, it is sufficient to now consider all states $s = (\gamma, 0)$ with $\gamma \leq b$ as accepting. Furthermore, noticing that all states $s = (\gamma, 0)$ such that $\gamma < 0$, $|\gamma| > (r - 1) \sum_{i=1}^{n} |a_i|$ and $|\gamma| > |b|$ can only lead to accepting states, and that all states $s = (\gamma, 0)$ such that $\gamma > 0$, $|\gamma| > (r - 1) \sum_{i=1}^{n} |a_i|$ and

$|\gamma| > |b|$ can only lead to nonaccepting states, these can be respectively collapsed into single accepting and nonaccepting states. The automaton thus only has a bounded number of states.

The backward construction can also be adapted in the following way: the states s for which the computed γ is not an integer or is not divisible by $\gcd(a_1, \ldots, a_n)$ are not discarded, but their value is rounded to the nearest lower integer γ' that is divisible by $\gcd(a_1, \ldots, a_n)$. This operation is correct since the sets of integer solutions of $\boldsymbol{a}.\boldsymbol{x} \leq \gamma$ and of $\boldsymbol{a}.\boldsymbol{x} \leq \gamma'$ are in this case identical. The resulting automaton is however no longer deterministic, but as was shown in [WB00], it can be determinized efficiently.

The construction of an automaton $\mathcal{A}_{\boldsymbol{a},b}$ that accepts all the infinite words $w \in \Sigma^\omega$ such that $0 \star w$ encodes a solution of $\boldsymbol{a}.\boldsymbol{x} \leq b$ that belongs to $[0,1]^n$ is again very similar to the one developed for equations in Section 4.1. The difference with the case of equations, in that we do not discard here the states $s = (\gamma, 0)$ for which the computed γ is greater than α'. Instead, we simply replace the value of γ by α', since the sets of solutions in $[0,1]^n$ of $\boldsymbol{a}.\boldsymbol{x} \leq \gamma$ and of $\boldsymbol{a}.\boldsymbol{x} \leq \alpha'$ are in this case identical. On the other hand, we still discard the states $s = (\gamma, 0)$ for which the computed γ is lower than α, since this implies that the inequation $\boldsymbol{a}.\boldsymbol{x} \leq \gamma$ has no solution in $[0,1]^n$. Notice again that this will produce a weak automaton that can be determinized and minimized.

5 Manipulating Sets of Integer and Real Vectors

Most applications of linear constraints require the possibility of transforming, combining, and checking represented sets according to various operators. For instance, deciding whether a formula ϕ of $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ is satisfiable can be done by first building the representations of the atoms of ϕ, which are equations and inequations (see Section 4), then applying the connectors and the quantifiers that appear in φ to these representations, and finally checking whether the resulting automaton (which recognizes exactly all solutions of ϕ) represents a non-empty set. The last step simply amounts to check that the language accepted by the finite-state representation is not empty. In this section, we give algorithms for applying Boolean connectors, quantifiers, and other specific operators to the finite-state representations of arithmetic sets.

5.1 Computing Boolean Combinations of Sets

Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ be automata representing in a base $r > 1$ (respectively) the sets $S_1, S_2, \ldots S_k \subseteq \mathbb{R}^n$. For each $i \in \{1, \ldots, k\}$, let $\mathcal{A}_i = (Q_i, \Sigma, \delta, Q_{0,i}, F_i)$, with $\Sigma = \{0, 1, \ldots, r-1, \star\}$, and let L_i denote the language accepted by $\mathcal{A}_i$. We assume that each $\mathcal{A}_i$ is weak, deterministic, and complete (meaning that for every state $q \in Q_i$ and symbol $a \in \Sigma$, there exists an outgoing transition from q labeled by a).

Let $S \subseteq \mathbb{R}^n$ be a Boolean combination $\mathcal{B}(S_1, S_2, \ldots, S_k)$ of the sets S_i, i.e., a set obtained by applying the operators $\cup$ (union), $\cap$ (intersection) and $\neg$ (complement) to the S_i.

In order to compute a finite-state representation of S, one first builds an automaton $\mathcal{A}$ that accepts the language $L = \mathcal{B}(L_1, L_2, \ldots, L_k)$. This consists of simulating the concurrent operation of the $\mathcal{A}_i$, accepting a word w whenever $\mathcal{B}(w \in L_1, w \in L_2, \ldots, w \in L_k)$ holds. Formally, we have $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, with

- $Q = Q_1 \times Q_2 \times \cdots \times Q_k$;
- $\delta : Q \times \Sigma \to Q : ((q_1, \ldots, q_k), a) \mapsto (\delta_1(q_1, a), \ldots, \delta_k(q_k, a))$;
- $Q_0 = Q_{0,1} \times \cdots \times Q_{0,k}$;
- $F = \{(q_1, \ldots, q_k) \in Q \mid \mathcal{B}(q_1 \in F_1, \ldots, q_k \in F_k)\}$.

(It is worth mentioning that this construction is valid only because the $\mathcal{A}_i$ are deterministic and weak. Every word corresponds to exactly one run for each $\mathcal{A}_i$, which ends up visiting only a group of states with the same accepting or nonaccepting status.)

The automaton $\mathcal{A}$ accepts an encoding w of a vector $\boldsymbol{x} \in \mathbb{R}^n$ if and only if $w \in \mathcal{B}(L_1, L_2, \ldots, L_k)$, i.e., if and only if $\boldsymbol{x} \in S$. This does not imply however that $\mathcal{A}$ forms a valid representation of S, since it may accept words that are not vector encodings. A suitable representation of S can therefore be obtained by intersecting $\mathcal{A}$ with an automaton accepting the set of all valid vector encodings, in other words, an automaton representing the set $\mathbb{R}^n$. Note that this intersection operation is only needed when L is not a subset of $L_1 \cup \cdots \cup L_k$, i.e., if $b_1 \vee b_2 \vee \ldots \vee b_k$ does not imply $\mathcal{B}(b_1, \ldots, b_k)$ for all Booleans b_i. Whenever required, the additional intersection operation can conveniently be performed as a part of the product construction described above. This construction preserves the determinism and weak nature of the automata.

5.2 Constructing Cartesian Products

Let $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, Q_{0,1}, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, Q_{0,2}, F_2)$ be weak deterministic finite-state representations of (respectively) the sets $S_1 \subseteq \mathbb{R}^{n_1}$ and $S_2 \subseteq \mathbb{R}^{n_2}$, with $n_1 > 0$ and $n_2 > 0$. A representation $\mathcal{A}$ of the Cartesian product $S = S_1 \times S_2$ is a finite-state machine that simulates repeatedly the operations of $\mathcal{A}_1$ for n_1 input symbols, and then those of $\mathcal{A}_2$ for the next n_2 symbols. The transitions labeled by the separator "$\star$" are handled in a special way by ensuring that they are followed at the same time in both automata. Let $n = n_1 + n_2$. Formally, we have $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, where

- $Q = Q_1 \times Q_2 \times \{0, 1, \ldots, n-1\}$;
- $\Sigma = \{0, 1, \ldots, r-1, \star\}$;
- $\delta : ((q_1, q_2, i), a) \mapsto \begin{cases} (\delta_1(q_1, a), q_2, i+1) & \text{if } i < n_1 \text{ and } a \not= \star, \\ (q_1, \delta_2(q_2, a), (i+1) \bmod n) & \text{if } i \geq n_1 \text{ and } a \not= \star; \\ (\delta_1(q_1, a), \delta_2(q_2, a), 0) & \text{if } a = \star. \end{cases}$
- $Q_0 = Q_{0,1} \times Q_{0,2} \times \{0\}$;
- $F = F_1 \times F_2 \times \{0, 1, \ldots, n-1\}$.

This construction preserves the determinism and the weak nature of the automata.

5.3 Applying Quantifiers

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be a finite-state representation of a set $S \subseteq \mathbb{R}^n$, with $n > 1$, and let $i \in \{1, \ldots, n\}$ be the index of a vector component. Quantifying existentially S with respect to the i-th vector component over the real domain amounts to projecting out this component, which yields the set

$$S' = \exists_i^{\mathbb{R}} S = \{(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \in \mathbb{R}^{n-1} \mid (\exists x_i \in \mathbb{R})((x_1, \ldots, x_n) \in S)\}.$$

In order to compute a representation of S', one needs to remove from $\mathcal{A}$ the transition labels corresponding to the i-th vector component. This is done by constructing a nondeterministic automaton $\mathcal{A}' = (Q', \Sigma, \delta', Q'_0, F')$, such that

- $Q' = Q \times \{0, 1, \ldots n-2\}$;
- $\Sigma = \{0, 1, \ldots, r-1, \star\}$;
- $\delta' : ((q,k), a) \mapsto \begin{cases} \{(\delta(q,a), (k+1) \bmod (n-1))\} & \text{if } k+1 \neq i \text{ and } a \neq \star, \\ \bigcup_{a' \in \{0,\ldots,r-1\}} \{(\delta(\delta(q,a'),a), (k+1) \bmod (n-1))\} & \\ \qquad \text{if } k+1 = i \text{ and } a \neq \star, & \\ \{(\delta(q,a), 0)\} & \text{if } a = \star; \end{cases}$
- $Q'_0 = \begin{cases} Q_0 \times \{0\} & \text{if } i \neq 1 \\ \bigcup_{a \in \{0,r-1\}, q_0 \in Q_0} \{(\delta(q_0, a), 0)\} & \text{if } i = 1; \end{cases}$
- $F' = F \times \{0\}$.

This approach introduces two problems. First, the resulting automaton $\mathcal{A}'$ is nondeterministic, which prevents its further manipulation by the algorithm outlined in Section 5.1 (especially when the set needs to be complemented). Second, although the language accepted by $\mathcal{A}'$ contains only valid encodings of vectors in $\exists_i S$, it may not accept all of them. (For instance, removing from a binary representation of the set $\{(10, 2)\}$ the transitions related to the first vector component would produce an automaton recognizing only the encodings of 2 that contain more than four digits in their integer part.) We address these two problems separately.

In order to determinize $\mathcal{A}'$, we exploit an interesting property of weak automata, which can directly be turned into co-Büchi automata: a weak Büchi automaton $(Q, \Sigma, \delta, Q_0, F)$ accepts the same language as the co-Büchi automaton $(Q, \Sigma, \delta, Q_0, Q \setminus F)$. As a consequence, weak automata can be determinized using the simple breakpoint construction [MH84, KV97] developed for co-Büchi automata. This construction proceeds as follows.

Let $\mathcal{A}' = (Q', \Sigma, \delta', Q'_0, F')$ be a weak non-deterministic co-Büchi automaton. The co-Büchi automaton $\mathcal{A}' = (Q'', \Sigma, \delta'', Q''_0, F'')$ defined as follows accepts the same language.

- $Q'' = 2^{Q'} \times 2^{Q'}$, the states of $\mathcal{A}''$ are pairs of sets of states of $\mathcal{A}'$;
- $Q''_0 = \{(Q'_0, \emptyset)\}$;
- For $(S, R) \in Q''$ and $a \in \Sigma$, the transition function is defined by
  - if $R = \emptyset$, then $\delta''((S,R), a) = (T, T \setminus F')$ where $T = \{q \mid \exists p \in S \text{ and } q \in \delta'(p, a)\}$, T is obtained from S as in the classical subset construction, and the second component of the pair of sets of states is obtained from T by eliminating states in F';

  - if $R \not= \emptyset$, then $\delta''((S,R),a) = (T, U \setminus F')$ where $T = \{q \mid \exists p \in S \text{ and } q \in \delta'(p,a)\}$, and $U = \{q \mid \exists p \in R \text{ and } q \in \delta'(p,a)\}$, the subset construction set is now applied to both S and R and states in F' are removed from U;
- $F' = 2^{Q'} \times \{\emptyset\}$.

When the automaton $\mathcal{A}''$ is in a state (S,R), R represents the states of $\mathcal{A}'$ that can be reached by a run that has not gone through a state in F' since that last "breakpoint", i.e. state of the form $(S,\emptyset)$. So, for a given word, $\mathcal{A}$ has a run that does not go infinitely often through a state in F' if and only if $\mathcal{A}''$ has a run that does not go infinitely often through a state in F''. Notice that the difficulty that exists for determinizing Büchi automata, which is to make sure that the *same* run repeatedly reaches an accepting state disappears since, for co-Büchi automata, we are just looking for a run that eventually avoids accepting states.

It is interesting to notice that the construction implies that all reachable states (S,R) of $\mathcal{A}''$ satisfy $R \subseteq S$. The breakpoint construction can thus be implemented as a subset construction in which the states in R are simply tagged. Experimental results (see Section 6) have shown that its practical efficiency is similar to that of the traditional subset construction for finite-word automata.

In general, determinizing a co-Büchi automaton does not produce a weak automaton. This problem can be alleviated, provided that the set of vectors undergoing the quantification operation (and thus the set obtained as a result) is definable in the theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. Indeed, it has been shown using topological arguments [BJW01] that any deterministic Büchi automaton recognizing such sets can easily be turned into a weak one by a simple operation. This operation actually consists of turning all the states belonging to the same strongly connected component of the automaton into accepting or non-accepting ones, depending on whether this component contains or not at least one accepting state.

We now address the problem of turning an automaton accepting some encodings of the vectors in a set into one that recognizes all of them. More precisely, in the present case, one needs to make sure that whenever the automaton accepts an encoding $u^k \cdot w$, where $u \in \{0, r-1\}^n$ is the sign prefix, it also accepts the words $u^j \cdot w$ for all j such that $1 < j < k$. In [Boi98], this problem is solved by computing for each possible sign prefix $u \in \{0, r-1\}^n$ the set of automaton states reachable after reading any word in u^*. These states are then made reachable from the initial state after reading any number of occurrences of u, thanks to a simple construction, and the process is repeated for other u.

The drawback of this approach is its systematic cost in $O(2^n)$, which limits is applicability to problems with a very small vector dimension. An improved algorithm has been developed [BL01], in which subsets of sign headers that are not distinguished by the automaton (in the sense that their reading always lead to the same automaton states) are handled collectively rather than separately. This algorithm has been implemented in the LASH tool [LAS], and experimental results have shown it to be of moderate practical cost even for large vector dimensions. Note that this automaton transformation may produce

non-determinism, and has thus to be performed prior to the determinization procedure discussed in this section.

We have only considered so far existential quantification over the reals. This generalizes naturally to universal quantification thanks to the complement operation described (as a particular instance of Boolean combination) in Section 5.1. In order to decide $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, one also needs to be able to quantify sets with respect to the integer domain, i.e., given a set $S \subseteq \mathbb{R}^n$ and a vector component index $i \in \{1, \ldots, n\}$, compute the sets

$$\exists_i^{\mathbb{Z}} S = \{(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \in \mathbb{R}^{n-1} \mid (\exists x_i \in \mathbb{Z})((x_1, \ldots, x_n) \in S)\}, \text{ and}$$
$$\forall_i^{\mathbb{Z}} S = \{(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \in \mathbb{R}^{n-1} \mid (\forall x_i \in \mathbb{Z})((x_1, \ldots, x_n) \in S)\}.$$

These operations are easily reduced to applying quantifiers over the reals and performing Boolean combinations, thanks to the following rules :

$$\exists_i^{\mathbb{Z}} S = \exists_i^{\mathbb{R}}(S \cap \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid x_i \in \mathbb{Z}\});$$
$$\forall_i^{\mathbb{Z}} S = \forall_i^{\mathbb{R}}(S \cup \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid x_i \not\in \mathbb{Z}\}).$$

Applying these rules requires an automaton that recognizes all vectors in which the i-th component is an integer (see Section 4).

5.4 Minimizing Finite-State Representations of Sets

Although the operations described in Sections 5.1 to 5.3 produce weak and deterministic automata, these can be unnecessarily large because of redundancies in their transition graph. In a recent paper [Löd01], it has been shown that weak deterministic automata admit a minimal form, unique up to isomorphism, which can be computed with $O(n \log n)$ cost. Sketchily, in order to minimize a weak deterministic automaton, one first locates the trivial strongly connected component in its transition graph (i.e., the components that do not contain cycles). Then, one modifies the accepting of non-accepting status of the states in these components (which does not affect the language accepted by the automaton) according to some specific rules. The result is then fed to the classical Hopcroft's algorithm [Hop71] for minimizing finite-state machines on finite words.

5.5 Other Operations

In verification applications, in order to explore infinite state-spaces, one needs to be able to compute infinite sets of reachable states in finite time. The concept of *meta-transition* has been introduced for this purpose in [BW94]. Intuitively, a meta-transition is associated to a cycle in the control graph of the analyzed system, and following it once leads to all the configurations that could be reached after following repeatedly that cycle any number of times.

If the program undergoing the analysis is based on integer and real variables, RVAs can be used as symbolic representations of its sets of configurations. If the operations performed by the program are definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, the computation of pre or post-images of such a set of configurations with respect to a given operation follows from the algorithms given in Sections 4 to 5.

In [Boi98], is has been shown that the effect of meta-transitions based on linear transformations and guards can be expressed in the theory $\langle \mathbb{Z}, +, \leq \rangle$ (i.e., in Presburger arithmetic), under some hypotheses. We summarize this result below.

Let $A \in \mathbb{Z}^{n \times n}$, $\boldsymbol{b} \in \mathbb{Z}^n$, $P \in \mathbb{Z}^{m \times n}$ and $\boldsymbol{q} \in \mathbb{Z}^m$, and let θ be the operation $P\boldsymbol{x} \leq \boldsymbol{q} \rightarrow \boldsymbol{x} := A\boldsymbol{x} + \boldsymbol{b}$, i.e., $\theta : \mathbb{R}^n \rightarrow \mathbb{R}^n : \boldsymbol{v} \mapsto A\boldsymbol{v} + \boldsymbol{b}$ if $P\boldsymbol{v} \leq \boldsymbol{q}$.

Theorem 1. *If there exists $p \geq 1$ such that*

- *The matrix A^p is diagonalizable, and*
- *Its eigenvalues belong to $\{0, 1\}$,*

then the closure $\theta^* = id \cup \theta \cup \theta^2 \cup \cdots$ of θ is definable in Presburger arithmetic.

The hypotheses of this theorem can be checked algorithmically, using only simple integer arithmetic [Boi98]. Its proof is constructive and turns into an algorithm for computing, for any set S in $\langle \mathbb{Z}, +, \leq \rangle$ the set $\theta^*(S)$ in terms of an expression defining S. Since $\langle \mathbb{Z}, +, \leq \rangle$ is a subtheory of $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, the same construction can also be carried out with RVAs.

The last operation considered in this study is the time-elapse transformation needed for the analysis of timed systems. Let $\boldsymbol{x} \in \mathbb{R}^n$ be a vector of clocks, the value of which evolves at the rate $\dot{\boldsymbol{x}}$ under the condition $\boldsymbol{a} \leq \dot{\boldsymbol{x}} \leq \boldsymbol{b}$ (in which $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}^n$ are constant vectors). Given a initial set $S \subseteq \mathbb{R}^n$ of clock values, the values reached after letting time elapse for an arbitrarily long period form the set

$$S' = \{\boldsymbol{x}' \in \mathbb{R}^n \mid (\exists \boldsymbol{x} \in S, t \in \mathbb{R}, \boldsymbol{\delta} \in \mathbb{R}^n)(t \geq 0 \wedge t\boldsymbol{a} \leq \boldsymbol{\delta} \leq t\boldsymbol{b} \wedge \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{\delta})\}.$$

Since this transformation is expressed in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, a RVA representing S' can easily be constructed from one representing S.

6 Examples and Experimental Results

As explained earlier, RVA have the interesting property of being able to represent both discrete and continuous features. As an example, consider the set

$$\{(x_1, x_2) \in \mathbb{R}^2 \mid (\exists x_3, x_4 \in \mathbb{R})(\exists x_5, x_6 \in \mathbb{Z})(x_1 = x_3 + 2x_5 \wedge x_2 = x_4 + 2x_6 \wedge 0 \leq x_3 \leq x_4 \leq 1)\}.$$

As illustrated in Figure 1, this set combines linear constraints and discrete periodicities. The minimal and deterministic RVA representing this set in base 2 is given in Figure 2, in its simultaneous form.

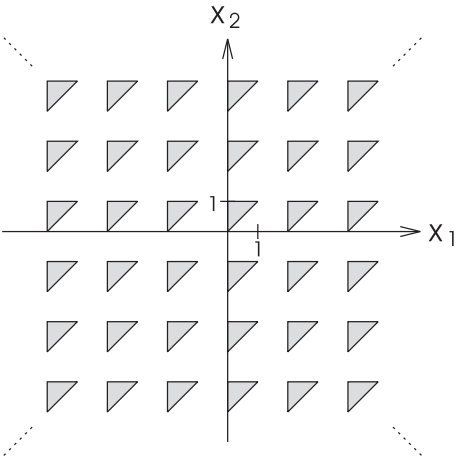


Fig. 1. A set with both continuous and discrete features

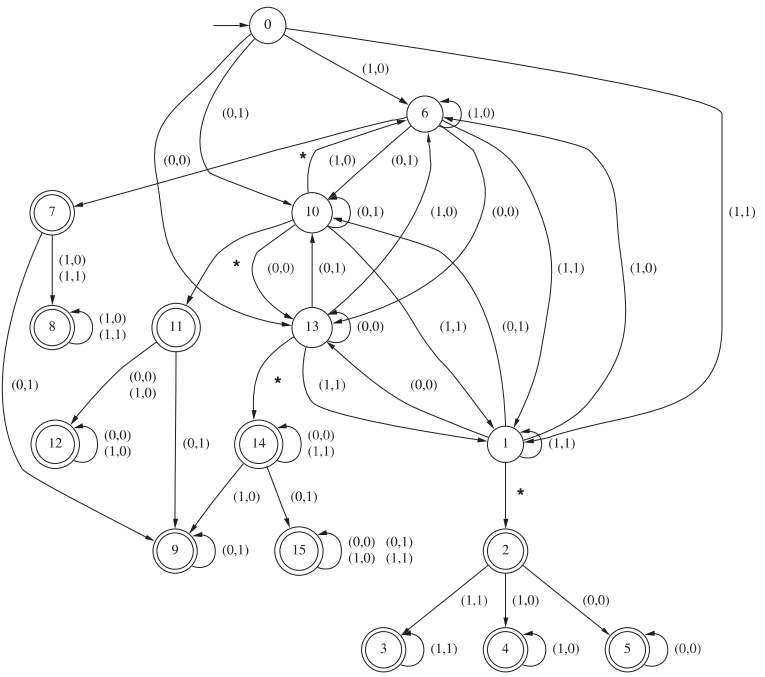


Fig. 2. RVA representing the set in Fig. 1

It has been mentioned in Section 5 that the cost of determinizing an RVA following a projection operation appears considerably smaller in practical applications than suggested by the worst-case complexity of the "breakpoint" construction. This observation is illustrated in Figure 3, in which the size of finite-state representations of sets of values obtained by combining linear constraints with arbitrary coefficients is given before and after undergoing the projection and determinization operations. The figure also compares the behaviors with respect to these operations of sets of real vectors represented by RVAs, and sets of integer solutions represented by automata on finite words (*Number Decision Diagrams, NDDs* [WB95]). As also substantiated by other experiments, RVAs seem just as usable in practice as NDDs.

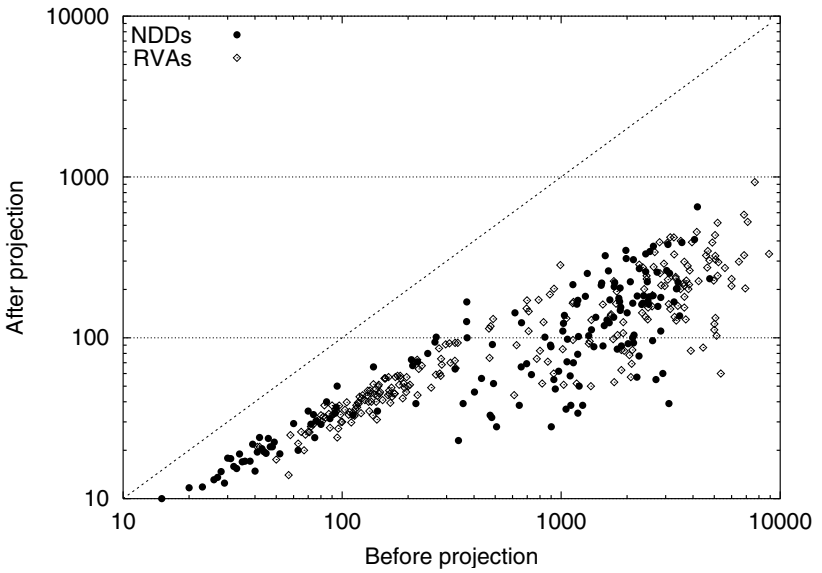


Fig. 3. Projecting and determinizing finite-state representations

7 Conclusions

This paper has overviewed automata-based techniques for handling linear constraints defined over the reals and integers. Though early experimental results are quite encouraging, there is little hope that these techniques will consistently outperform more traditional approaches when these can be applied. The reason for this is that the automaton one computed for a formula contains a lot of explicit information that might not be needed, for instance when just checking satisfiability. Computing this unneeded information comes at a cost that can be avoided by other approaches.

On the other hand, while the information made explicit by the automaton can be used repeatedly, there can be a substantial advantage to having it available: though the initial computation of the automaton will be costly it will speed up subsequent computations. This can for instance be the case in the symbolic verification of infinite-state systems [WB98].

A major advantage of the automaton-based approach is that it handles traditionally harder to deal with cases, such as nonconvex sets, just as simply as the more usual convex sets. Furthermore, it provides a normal form for represented sets in all cases, the absence of which in other approaches is a recurrent problem, for example in the case of nonconvex sets. Finally, the fact that it can handle constraints defined over the integers and reals combined is a significant new capability. An application that has already been considered is the verification of hybrid systems [BBR97], but there are probably also many others.

Acknowledgement

Sébastien Jodogne has contributed substantially to the implementation of the arithmetic part of the LASH tool and has done a careful reading of this paper.

References

[BBR97] B. Boigelot, L. Bronne, and S. Rassart. An improved reachability analysis method for strongly linear hybrid systems. In *Proc. 9th Int. Conf.on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 167–178, Haifa, June 1997. Springer-Verlag. 2, 18

[BC96] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proceedings of CAAP'96*, number 1059 in Lecture Notes in Computer Science, pages 30–43. Springer-Verlag, 1996. 2, 6

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990. 2

[BJW01] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. On the use of weak automata for deciding linear arithmetic with integer and real variables. In *Proc. International Joint Conference on Automated Reasoning (IJCAR)*, volume 2083 of *Lecture Notes in Computer Science*, pages 611–625, Siena, June 2001. Springer-Verlag. 2, 5, 13

[BL01] B. Boigelot and L. Latour. Counting the solutions of Presburger equations without enumerating them. In *Proc. International Conference on Implementations and Applications of Automata*, Lecture Notes in Computer Science, Pretoria, South Africa, July 2001. Springer-Verlag. To appear. 13

[Boi98] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1998. 2, 13, 15

[BRW98] Bernard Boigelot, Stéphane Rassart, and Pierre Wolper. On the expressiveness of real and integer arithmetic automata. In *Proc. 25th Colloq. on Automata, Programming, and Languages (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 152–163. Springer-Verlag, July 1998. 5, 6, 8

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. 2

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift Math. Logik und Grundlagen der Mathematik*, 6:66–92, 1960. 2

[BW94] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In *Computer Aided Verification, Proc. 6th Int. Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67, Stanford, California, June 1994. Springer-Verlag. 14

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. *Theory of Machines and Computation*, pages 189–196, 1971. 8, 14

[KV97] O. Kupferman and M. Vardi. Weak alternating automata are not that weak. In *Proc. 5th Israeli Symposium on Theory of Computing and Systems*, pages 147–158. IEEE Computer Society Press, 1997. 12

[LAS] The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`. 2, 13

[Löd01] C. Löding. Efficient minimization of deterministic weak $\omega-$automata, 2001. Submitted for publication. 2, 14

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984. 12

[MS97] O. Maler and L. Staiger. On syntactic congruences for ω-languages. *Theoretical Computer Science*, 183(1):93–112, 1997. 2

[MSS86] D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *Proc. 13th Int. Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1986. 2, 4

[Pug92] W. Pugh. A practical algorithm for exact array dependency analysis. *Comm. of the ACM*, 35(8):102, August 1992. 1

[Sta83] L. Staiger. Finite-state ω-languages. *Journal of Computer and System Sciences*, 27(3):434–448, 1983. 2

[Tho90] Wolfgang Thomas. Automata on infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science – Volume B: Formal Models and Semantics*, chapter 4, pages 133–191. Elsevier, Amsterdam, 1990. 2, 4

[WB95] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32, Glasgow, September 1995. Springer-Verlag. 17

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer-Verlag. 1, 18

[WB00] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*,

volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag. 2, 6, 10

Logic Databases on the Semantic Web: Challenges and Opportunities

Stefan Decker

Department of Computer Science, Stanford University
Gates Hall 4A, Stanford, CA 94305-9040 USA
stefan@db.stanford.edu

Abstract. An extended version of the abstract is available at
http://www.SemanticWeb.org/ICLP2002

1 Introduction

Until now, the Web has mainly designed for direct human consumption. The next step in the evolution, dubbed the "Semantic Web", aims at machine-processable information, enabling intelligent services such as information brokers, search agents, information filters, and direct B2B communication, which offers greater functionality and interoperability than the current stand-alone services.

> *"The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation. The first steps in weaving the Semantic Web into the structure of the existing Web are already under way. In the near future, these developments will usher in significant new functionality as machines become much better able to process and "understand" the data that they merely display at present".* [1]

Several Working groups within the W3C are working on standards for knowledge representation languages, which help to make this dream come true, e.g., the Web Ontology Working Group[1] and the RDF Core Working Group[2].

This development creates opportunities and challenges for logic databases:

- The presence of vast amounts of machine-processable, reusable data requires means for declarative data processing, since explicit programming is no longer economically feasible. In the Semantic Web many different communities are publishing their formal data, and it is unlikely that established data models for representing this data will disappear. Examples of already established data models include UML, TopicMaps, RDF Schema, Entity Relationship Models, DAML+OIL, and more, highly specialized data models. Integrating data based on these different data models has proven to be an

[1] See http://www.w3.org/2001/sw/WebOnt/
[2] See http://www.w3.org/2001/sw/RDFCore/

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 20–21, 2002.
© Springer-Verlag Berlin Heidelberg 2002

error-prone and expensive task: different storage and query engines have to be combined into one program, and data has to be translated constantly from one representation to another. Rules languages should help to overcome these problems.
- Rule processing subsystems will become (and actually are already) more and more standard parts of business applications. This creates the need to establish a rule exchange standard[3], which, once established, helps to exchange reusable rule definitions. This standard would not only help to exchange rules, but also provides a standard for rule-based knowledge representation on the Web.

The above mentioned points create technical, unsolved problems. Languages need to be developed which allow to specify data transformations for various data models. Efficient query and inference techniques have to be developed. Technologies developed by the Logic Programming and Deductive Database Community can help to built the Semantic Web by overcoming these problems.

References

1. Berners-Lee, T., Hendler, J., Lassila, O.,: The Semantic Web. In: Scientific American, May 2001, http://www.sciam.com/2001/0501issue/0501berners-lee.html 20
2. Boley, H., Tabet, S., Wagner, G.: Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In *International Semantic Web Working Symposium (SWWS)*, 2001, http://www.SemanticWeb.org/SWWS 21

[3] See e.g., RuleML [2] at http://www.dfki.uni-kl.de/ruleml/

An Abductive Approach for Analysing Event-Based Requirements Specifications

Alessandra Russo[1], Rob Miller[2], Bashar Nuseibeh[3], and Jeff Kramer[1]

[1] Imperial College of Science, Technology and Medicine
London SW7 2BT, U.K.
{ar3,jk}@ic.ac.uk
[2] University College London
London WC1E 6BT, U.K.
rsm@ucl.ac.uk
[3] The Open University, Walton Hall
Milton Keynes, MK7 6AA, U.K.
B.A.Nuseibeh@open.ac.uk

Abstract. We present a logic and logic programming based approach for analysing event-based requirements specifications given in terms of a system's reaction to events and safety properties. The approach uses a variant of Kowalski and Sergot's Event Calculus to represent such specifications declaratively and an abductive reasoning mechanism for analysing safety properties. Given a system description and a safety property, the abductive mechanism is able to identify a complete set of counterexamples (if any exist) of the property in terms of symbolic "current" states and associated event-based transitions. A case study of an automobile cruise control system specified in the SCR framework is used to illustrate our approach. The technique described is implemented using existing tools for abductive logic programming.

1 Introduction

Requirements specification analysis is a critical activity in software development. Specification errors, which if undetected often lead to system failures, are in general less expensive to correct than defects detected later in the development process. This paper describes a formal approach to the detection and analysis of errors, and an associated logic programming tool, that have the following two desirable characteristics. First, the tool is able to verify some properties and detect some errors even when requirements specifications are only partially completed, and even when only partial knowledge about the domain is available. In particular, our approach does not rely on a complete description of the initial state(s) of the system, making it applicable to systems embedded in complex environments whose initial conditions cannot be completely predicted. Second, the tool provides "pinpoint" diagnostic information about detected errors (e.g. violated safety properties) as a "debugging" aid for the engineer. In practical terms, it

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 22–37, 2002.
© Springer-Verlag Berlin Heidelberg 2002

is the integration of both characteristics that distinguishes our approach from other formal techniques, such as model checking or theorem proving [6].

Our focus is on event-based requirements specifications. In this paper, we will regard such specifications as composed of system descriptions, i.e. expressed in terms of required reactions to events (inputs, changes in environmental conditions, etc.), and global system invariants. For simplicity we restrict our attention to "single-state" invariants (e.g. safety properties) although we speculate that our approach could be adapted for other types of property.

The approach uses the Event Calculus (EC) [17] to declaratively model event-based requirements specifications. The choice of EC is motivated by both practical and formal needs, and gives several advantages. First, in contrast to pure state-transition representations, the EC ontology includes an explicit time structure that is independent of any (sequence of) events under consideration. This characteristic makes it straightforward to model event-based systems where a number of input events may occur simultaneously, and where the system behavior may in some circumstances be non-deterministic (see [22]). Second, the EC ontology is close enough to existing types of event-based requirements specifications to allow them to be mapped automatically into the logical representation. This allows our approach and tool to be used as a "back-end" to existing requirements engineering representational methods. Both the semantics of the front-end specification language and individual specifications themselves can be represented in EC. Third, we can prove a general property of the particular class of EC representations employed here which allows us to reason with a reduced "two-state" representation (see Section 2.3), thus substantially improving the efficiency of our tool. Fourth, we can build on a substantial body of existing work in applying abductive reasoning techniques to EC representations [16,22].

This brings us to the second corner stone of our approach – the use of abduction. Abduction has already proved suitable for automating knowledge-based software development [20,27]. Our approach employs abduction in a refutation mode to verify global system invariants with respect to event-based system descriptions. Given a system description and an invariant, the abduction is able to identify a complete set of counterexamples (if any exist) to the system invariant, where each counterexample is in terms of a "current" system state and an associated event-based transition. Failure to find a counterexample establishes the validity of the invariant with respect to the system description. (Thus, in A.I. terminology, each counterexample is an "explanation" for the "observation" which is the negation of the invariant at an arbitrary symbolic time-point.) The particular form of these counterexamples makes them ideal as diagnoses which can be used to modify the specification appropriately, by altering either the event-based system description, or the set of global system invariants, or both.

The abductive decision procedure employed by our approach has several desirable features. It always terminates, in contrast to most conventional theorem proving techniques. It does not rely on a complete description of an initial state (in contrast to model-checking approaches). Like the EC representation, it sup-

ports reasoning about specifications whose state-spaces may be infinite. This last feature is mainly because the procedure is goal- or property-driven.

The next section describes our general approach. It is followed by an illustrative case study involving analysis of an SCR tabular specification. We conclude with some remarks about related and future work.

2 Our Approach

As stated above, we will regard requirements specifications as composed of system descriptions and global system invariants. The analysis task that we are concerned with is to discover whether a given system description satisfies all system invariants, and if not why not. We express a collection of system invariants as logical sentences $I_1, \ldots, I_n$ and an event-based system description as a set of rules S. Thus for each system invariant I_i, we need to evaluate whether $S \models I_i$, and to generate appropriate diagnostic information if not. The Event Calculus representation we have employed allows us to use an abductive reasoning mechanism to combine these two tasks into a single automated decision procedure.

2.1 Abduction for Verification

Abduction is commonly defined as the problem of finding a set of hypotheses (an "explanation" or "plan") of a specified form that, when added to a given formal specification, allows an "observation" or "goal" sentence to be inferred, without causing contradictions [16]. In logical terms, given a domain description D and a sentence (goal) G, abduction attempts to identify a set Δ of assertions such that $(D \cup \Delta) \models G$ and $(D \cup \Delta)$ is consistent. The set Δ must consist only of *abducible* sentences, where the definition of what is abducible is generally domain-specific. Δ is often required to be minimal.

From a computational view, abductive procedures (i.e. procedures to find Δ's) are usually composed of two phases, an *abductive phase* and a *consistency phase*, that interleave with each other. Each abducible generated during the first phase is temporarily added to a set of abducibles that have already been generated. But this addition is only made permanent if the second phase confirms that the entire new set of abducibles is consistent with the specification. Furthermore, the abducibles together with the system description often have to satisfy a given set of *integrity constraints*. In general this (re)checking for consistency and satisfaction of constraints can be computationally expensive, but the particular form of our EC specifications together with a theoretical result regarding plan consistency in [21] allows us to avoid such pitfalls.

In our abductive approach the problem of proving that, for some invariant I_i, $D \models I_i$ is translated into an equivalent problem of showing that it is not possible to consistently extend D with assertions that particular events have actually occurred (i.e. with a Δ) in such a way that the extended description entails $\neg I_i$. In other words, there is no set Δ such that $D \cup \Delta \models \neg I_i$. The equivalence

of these problems is dependent on the particular Event Calculus representation used (see [26]). We solve the latter problem by attempting to generate such a Δ using a complete abductive decision procedure, and refer to this process as using abduction in a *refutation mode*. If the procedure finds a Δ then the assertions in Δ act as a counterexample. As we shall see, the form of such counterexamples makes them ideal as diagnostic information that can be utilised to change the description and/or invariants. The counterexamples that our approach generates describe particular events occurring in particular "contexts" (i.e. classes of "current states"). To be relevant, these contexts must themselves satisfy the invariants. This is ensured by considering the invariants as integrity constraints on a symbolic current state, which prunes the set of possible counterexamples. A detailed description of the particular abductive proof procedure used in our approach can be found in [14].

2.2 The Event Calculus

The Event Calculus is a logic-based formalism for representing and reasoning about dynamic systems. Its ontology includes an explicit structure of time independent of any (sequence of) events or actions under consideration. As we shall see, this characteristic makes it straightforward to model a wide class of event-driven systems including those that are non-deterministic, those in which several events may occur simultaneously, and those for which the state space is infinite. Our approach has, so far, been tested only on specifications for deterministic systems, such as the case study described in Section 3. However, we are currently investigating its applicability to LTS style specifications [18], which may be for concurrent and non-deterministic systems.

Our approach adapts a simple classical logic form of the EC [22], whose ontology consists of (i) a set of *time-points* isomorphic to the non-negative integers, (ii) a set of time-varying properties called *fluents*, and (iii) a set of *event types* (or *actions*). The logic is correspondingly sorted, and includes the predicates *Happens*, *Initiates*, *Terminates* and *HoldsAt*, as well as some auxiliary predicates defined in terms of these. $Happens(a, t)$ indicates that event (or action) a actually occurs at time-point t. $Initiates(a, f, t)$ (resp. $Terminates(a, f, t)$) means that if event a were to occur at t it would cause fluent f to be true (resp. false) immediately afterwards. $HoldsAt(f, t)$ indicates that fluent f is true at t. So, for example $[Happens(A_1, T_4) \wedge Happens(A_2, T_4)]$ indicates that events A_1 and A_2 occur simultaneously at time-point T_4.

System Descriptions as Axiomatisations Every EC description includes a core collection of domain-independent axioms that describe general principles for deciding when fluents hold or do not hold at particular time-points. In addition, each specification includes a collection of domain-dependent sentences, describing the particular effects of events or actions (using the predicates *Initiates* and *Terminates*), and may also include sentences stating the particular time-points at which instances of these events occur (using the predicate *Happens*).

It is convenient to introduce two auxiliary predicates, *Clipped* and *Declipped*. $Clipped(T_1, F, T_2)$ means that some event occurs between the times T_1 and T_2 which terminates the fluent F:

$$Clipped(t_1, f, t_2) \stackrel{\text{def}}{\equiv} \exists a, t[Happens(a, t) \wedge t_1 \leq t < t_2 \wedge Terminates(a, f, t)] \quad \text{(EC1)}$$

(In all axioms all variables are assumed to be universally quantified with maximum scope unless otherwise stated.) Similarly, $Declipped(T_1, F, T_2)$ means that some event occurs between the times T_1 and T_2 which initiates the fluent F:

$$Declipped(t_1, f, t_2) \stackrel{\text{def}}{\equiv} \exists a, t[Happens(a, t) \wedge t_1 \leq t < t_2 \wedge Initiates(a, f, t)] \quad \text{(EC2)}$$

Armed with this notational shorthand, we can state the three general (commonsense) principles that constitute the domain-independent component of the EC: (i) fluents that have been initiated by event occurrences continue to hold until events occur that terminate them:

$$HoldsAt(f, t_2) \leftarrow [Happens(a, t_1) \wedge Initiates(a, f, t_1) \wedge t_1 < t_2 \wedge \neg Clipped(t_1, f, t_2)] \quad \text{(EC3)}$$

(ii) fluents that have been terminated by event occurrences continue not to hold until events occur that initiate them:

$$\neg HoldsAt(f, t_2) \leftarrow [Happens(a, t_1) \wedge Terminates(a, f, t_1) \wedge t_1 < t_2 \wedge \neg Declipped(t_1, f, t_2)] \quad \text{(EC4)}$$

(iii) fluents only change status via occurrence of initiating or terminating events:

$$HoldsAt(f, t_2) \leftarrow [HoldsAt(f, t_1) \wedge t_1 < t_2 \wedge \neg Clipped(t_1, f, t_2)] \quad \text{(EC5)}$$

$$\neg HoldsAt(f, t_2) \leftarrow [\neg HoldsAt(f, t_1) \wedge t_1 < t_2 \wedge \neg Declipped(t_1, f, t_2)] \quad \text{(EC6)}$$

To illustrate how the effects of particular events may be described in the domain-dependent part of a specification using *Initiates* and *Terminates*, we will describe an electric circuit consisting of a single light bulb and two switches A and B all connected in series. We need three fluents, *SwitchAOn*, *SwitchBOn* and *LightOn*, and two actions *FlickA* and *FlickB*. We can describe facts such as (i) that flicking switch A turns the light on, provided that switch A is not already on and that switch B is already on (i.e. connected) and is not simultaneously flicked, (ii) that if neither switch is on, flicking them both simultaneously causes the light to come on, and (iii) that if either switch is on, flicking it causes the

light to go off (irrespective of the state of the other switch):

$$\begin{aligned}
&Initiates(FlickA, LightOn, t) \leftarrow [\neg HoldsAt(SwitchAOn, t) \\
&\qquad\qquad \wedge HoldsAt(SwitchBOn, t) \wedge \neg Happens(FlickB, t)] \\
&Initiates(FlickA, LightOn, t) \leftarrow [\neg HoldsAt(SwitchAOn, t) \\
&\qquad\qquad \wedge \neg HoldsAt(SwitchBOn, t) \wedge Happens(FlickB, t)] \\
&Terminates(FlickA, LightOn, t) \leftarrow HoldsAt(SwitchAOn, t)
\end{aligned}$$

$$Terminates(FlickB, LightOn, t) \leftarrow HoldsAt(SwitchBOn, t)$$

In fact, in this example we need a total of five such sentences to describe the effects of particular events or combinations of events on the light, and a further four sentences to describe the effects on the switches themselves. Although for readability these sentences are written separately here, it is the *completions* (i.e. the if-and-only-if transformations) of the sets of sentences describing *Initiates* and *Terminates* that are actually included in the specification (see [22] for details). The use of completions avoids the frame problem, i.e. it allows us to assume that the only effects of events are those explicitly described.

For many applications, it is appropriate to include similar (completions of) sets of sentences describing which events occur (when using the predicate *Happens*). However, in this paper we wish to prove properties of systems under all possible scenarios, i.e. irrespective of which events actually occur. Hence our descriptions leave *Happens* undefined, i.e. they allow models with arbitrary interpretations for *Happens*. In this way, we effectively simulate a branching time structure that covers every possible series of events. In other words, by leaving *Happens* undefined we effectively consider, in one model or another, every possible path through a state-transition graph.

2.3 Efficient Abduction with Event Calculus

In this paper, we wish to take an EC description such as that above and use it to test system invariants. In the language of the EC these are expressions involving *HoldsAt* and universally quantified over time, such as $\forall t.[HoldsAt(SwitchAOn, t) \vee \neg HoldsAt(LightOn, t)]$. It is (potentially) computationally expensive to prove such sentences by standard (deductive or abductive) theorem-proving. To overcome this problem we have reduced this inference task to a simpler one as stated by the following theorem.

Theorem 1. *Let EC*($\mathbb{N}$) *be an Event Calculus description with time-points interpreted as the natural numbers* $\mathbb{N}$*, and let* $\forall t.I(t)$ *be an invariant. Let* $\mathbb{S}$ *be the time structure consisting of two points* S_c *and* S_n*, with* $S_c < S_n$*. Then EC*($\mathbb{N}$) $\models \forall t.I(t)$ *if and only if EC*($\mathbb{N}$) $\models I(0)$ *and EC*($\mathbb{S}$) $\cup\, I(S_c) \models I(S_n)$*. (Proof by induction over* $\mathbb{N}$*, see [26].)*

Hence to show for some invariant $\forall t.I(t)$ that $EC(\mathbb{N}) \models \forall t.I(t)$ it is sufficient to consider only a symbolic time-point S_c and its immediate successor S_n ("c"

for "current" and "n" for "next"), assume the invariant to be true at S_c, and demonstrate that its truth then follows at S_n. Theorem 1 is applicable even when complete information about the initial state of the system is not available. Its utilisation reduces computational costs considerably because, in the context of $EC(\mathbb{S})$, it allows us to re-write all our EC axioms with ground time-point terms. For example, (EC5) becomes:

$$HoldsAt(f, S_n) \leftarrow [HoldsAt(f, S_c) \wedge \neg Clipped(S_c, f, S_n)]$$

Once the EC representation of an event-based requirements specification is provided (perhaps by automatic translation), the approach applies existing abductive tools to analyse this specification. Using the reduced time structure described above, our approach proves assertions of the form $EC(\mathbb{S}) \cup I(S_c) \models I(S_n)$ by showing that a *complete abductive procedure* fails to produce a set Δ of *HoldsAt* and *Happens* facts (grounded at S_c) such that $EC(\mathbb{S}) \cup I(S_c) \cup \Delta \models \neg I(S_n)$. This procedure is valid given the particular form of the EC descriptions and under reasonable assumption that only a finite number of events can occur in a given instant. Theorem 1 then allows us to confirm that, provided $I(0)$ is true, $\forall t.I(t)$ is also true. If on the other hand the abductive procedure produces such a set Δ, then this Δ is an explicit indicator of where in the specification there is a problem. The case study gives such an example of generation of diagnostic information from the violation of invariants.

The particular form of our EC system descriptions allows us to further reduce computational costs by largely avoiding the consistency checking normally associated with abduction. This is because it ensures that any internally consistent, finite collection of *Happens* literals is consistent with any related description. Therefore, it is necessary only to check the consistency of candidate *HoldsAt* literals against the system invariants, and this can be done efficiently because both these types of expression are grounded at S_c.

Logic Programming Implementation Page limitations prevent us from describing in detail the implementation of our abductive tool. However, it is implemented in Prolog, using a simplified version of the abductive logic program module described in [14]. The logic program conversion of the given (classical logic) Event Calculus specification is achieved using the method described in [15], which overcomes the potential mismatch between the negation-as-failure used in the implementation and the classical negation used in the specification.

We have been able to formally prove the correctness of our Prolog tool with respect to the theoretical framework described in this paper, and this is fully documented in [26]. Because we are using abduction in what we have described as "refutation mode", the proof relies on demonstrating both the soundness and completeness of the Prolog abductive computation w.r.t. the classical logic description of abduction (at least in the context of the EC axiomatisation) described here. The proof of completeness builds on the work in [13] on a generalised stable model semantics for abduction, and is valid for a well-defined class of deterministic EC domain descriptions.

3 A Case Study

In this section we describe, via an example, an application of our approach to analysing Software Cost Reduction (SCR) specifications. We show how our tool analyses particular SCR-style system invariants, called *mode invariants*, with respect to event-based system descriptions expressed as SCR *mode transition tables*. The SCR approach has been proven useful for expressing the requirements of a wide range of large-scale real-world applications [1,7,10,23] and is an established method for specifying and analysing event-based systems.

3.1 SCR Specifications

The SCR method is based on Parnas's "Four Variable Model", which describes a required system's behavior as a set of mathematical relations between *monitored* and *controlled* variables, and input and output data items [25]. Monitored variables are environmental entities that influence the system behavior, and controlled variables are environmental entities that the system controls. For simplicity, our case study uses only Boolean variables. (Non-Boolean variables can always be reduced to Boolean variables, i.e. predicates defined over their values.) SCR facilitates the description of natural constraints on the system behavior, such as those imposed by physical laws, and defines system requirements in terms of relations between monitored and controlled variables, expressed in tabular notation.

Predicates representing monitored and controlled variables are called *conditions* and are defined over single system states. An *event* occurs when a system component (e.g, a monitored or controlled variable) changes value. Full SCR specifications can include *mode transition*, *event* and *condition* tables to describe a required system behavior, *assertions* to define properties of the environment, and *invariants* to specify properties that are required to always hold in the system (see [4,9,10]). However, this case study concerns a simple SCR specification consisting of just a single mode transition table and a list of system invariants.

Mode Transition Tables *Mode classes* are abstractions of the system state space with respect to monitored variables. Each mode class can be seen as a state machine, defined on the monitored variables, whose states are modes and whose transitions, called *mode transitions*, are triggered by changes on the monitored variables. Mode transition tables represent mode classes and their respective transitions in a tabular format. The mode transition table for our case study, taken from [3], is given in Table 1. It is for an automobile cruise control system. Note that the table already reflects basic properties of monitored variables. For example, the two transitions from "Inactive" to "Cruise" take into account the environmental property that in any state a cruise control lever is in exactly one of the three positions "Activate", "Deactivate" or "Resume". So, for example, whenever "Activate" changes to true, either "Deactivate" or "Resume" changes to false. For a more detailed description of this case study see [3].

Mode transition events occur when one or more monitored variables change their values. Events are of two types: "@T(C)" when a condition C changes from false to true, and "@F(C)" when C changes from true to false. C is called a *triggered condition*. For example, in the automobile cruise control system the event "@T(Ignited)" denotes that the engine of the automobile has changed from not being ignited to being ignited. Event occurrences can also depend on the truth/falsity of other conditions. In this case, they are called *conditioned events*. For example, in Table 1 the mode transition defined in the second row is caused by the occurrence of conditioned event "@F(Ignited)" whose condition is that "Running" is false. Different semantics have been used for conditioned events [11], all of which are expressible in our Event Calculus approach. In this case study, we have adopted the following interpretation. An event "@T(C)" conditional on "D" means that "C" is false in the current mode and is changed to true in the new mode, while "D" is true in the current mode and stays true in the new mode. The interpretation is similar for an event "@F(C)" conditional on "D", but with "C" changing truth value from true to false. In a mode transition table, each row is a transition from a current mode, indicated in the left most column of the table, to a new mode, specified in the right most column. The central part of the table defines the events that cause the transition. A triggered event "C" can have entries equal to "@T" or "@F". Monitored variables that are conditions for the occurrence of an event can have entry equal to "t" or "f". Monitored variables that are irrelevant for the transition have a "-" entry.

Table 1. Mode transition table for an automobile cruise control system

Current Mode	**Ignited**	**Running**	**Toofast**	**Brake**	**Acti-vate**	**Deactivate**	**Resume**	**New Mode**
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	f	-	-	-	-	-	Off
	@F	@F	-	-	-	-	-	
	t	t	-	f	@T	@F	f	Cruise
	t	t	-	f	@T	f	@F	
Cruise	@F	@F	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	-	@T	-	-	-	-	
	t	t	f	@T	-	-	-	Override
	t	t	f	-	@F	@T	f	
	t	t	f	-	f	@T	@F	
Override	@F	@F	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	t	-	f	@T	@F	f	Cruise
	t	t	-	f	@T	f	@F	
	t	t	-	f	f	@F	@T	
	t	t	-	f	@F	f	@T	

SCR mode transition tables can be seen as shorthand for much larger tables in two respects. First, a "-" entry for a condition in the table is shorthand for any of the four possible condition entries "@T", "@F", "t" and "f". This means that any transition between a current and new mode specified in a table using n dashes is in effect shorthand for up to 4^n different transitions, between the same current and new modes, given by the different combinations of entries for each of the dashed monitored variables. For instance, the first transition in Table 1 from "Inactive" to "Cruise" is shorthand for four different transitions between "Inactive" and "Cruise" given, respectively, by each of the four entries "t", "f", "@T" and "@F" for the condition "Toofast". Second, tables are made much more concise by the non-specification of transitions between identical modes. A table basically describes a function that defines, for each current mode and each combination of condition values, a next mode of the system. This next mode may or may not be equal to the current mode. The function thus uniquely captures the system requirements. However in specifying real system behavior only the transitions between current and next modes that are different are explicitly represented in SCR tables. The other "transitions" (where current and next modes are identical) are implicit and thus omitted or "hidden" from the table. Hence we may regard the meaning of real SCR mode transition tables as being given by "full extended" (and very long!) mode transition tables which do not utilise "-" dashes and include a row (which might otherwise be "hidden" in the sense described above) for each possible combination of current mode and "t", "f", "@T" and "@F" condition entries. Both the implicit "hidden rows" and the dashes need to be taken into account when analysing invariants with respect to the real (concise) version of an SCR mode transition table. Our case study shows that both can indeed be causes for mismatch between SCR tables and system invariants, as they may obscure system behaviors that violate these invariants.

Mode Invariants Mode invariants are unchanging properties (specification assertions) of the system regarding mode classes, which should be satisfied by the system specification. In our case study of an automobile cruise control system, an example of an invariant is $[Cruise \rightarrow (Ignited \wedge Running \wedge \neg Brake)]$. This means that whenever the system is in mode "Cruise", the conditions "Ignited" and "Running" must be true and "Brake" must be false. In SCR notation mode invariants are formulae of the form $m \rightarrow P$, where m is a mode value of a certain mode class and P is a logical proposition over the conditions used in the associated mode transition table. A mode transition table of a given mode class has to satisfy the mode invariants related to that mode class.

3.2 Abductive Analysis of Invariants

The Translation We can now illustrate the use of our abductive EC approach to analysing mode invariants in SCR mode transition tables. In our translation, both conditions and modes are represented as fluents, which we will refer to as *condition fluents* and *mode fluents* respectively. Although in reality many different types of external, real-word events may affect a given condition, SCR tables

abstract these differences away and essentially identify only two types of events for each condition - a "change-to-true" (@T) and a "change-to-false" (@F) event. Hence in our EC translation there are no independent event constants, but instead two functions $@T$ and $@F$ from fluents to events, and for each condition fluent C, the two axioms:

$$Initiates(@T(C), C, t) \quad \text{(S1)}$$
$$Terminates(@F(C), C, t) \quad \text{(S2)}$$

The translation of tables into EC axioms (rules) is modular, in that a single *Initiates* and a single *Terminates* rule is generated for each row. For a given row, the procedure for generating the *Initiates* rule is as follows. The *Initiates* literal in the left-hand side of the rule has the new mode (on the far right of the row) as its fluent argument, and the first @T or @F event (reading from the left) as its event argument. The right-hand side of the rule includes a *HoldsAt* literal for the current mode and a pair of *HoldsAt* and *Happens* literals for each "non-dash" condition entry in the row. Specifically, if the entry for condition C is a "t" this pair is $HoldsAt(C, t) \wedge \neg Happens(@F(C), t)$, for "f" it is $\neg HoldsAt(C, t) \wedge \neg Happens(@T(C), t)$, for "@T" it is $\neg HoldsAt(C, t) \wedge Happens(@T(C), t)$, and for "@F" it is $HoldsAt(C, t) \wedge Happens(@F(C), t)$. The *Terminates* rule is generated in exactly the same way, but with the current mode as the fluent argument in the *Terminates* literal. For example, the seventh row in Table 1 is translated as follows:

$$\begin{aligned} Initiates(@F(Running), Inactive, t) \leftarrow\ & [HoldsAt(Cruise, t) \\ & \wedge HoldsAt(Ignited, t) \wedge \neg Happens(@F(Ignited), t) \\ & \wedge HoldsAt(Running, t) \wedge Happens(@F(Running), t)] \end{aligned}$$

$$\begin{aligned} Terminates(@F(Running), Cruise, t) \leftarrow\ & [HoldsAt(Cruise, t) \\ & \wedge HoldsAt(Ignited, t) \wedge \neg Happens(@F(Ignited), t) \\ & \wedge HoldsAt(Running, t) \wedge Happens(@F(Running), t)] \end{aligned}$$

Clearly, this axiom pair captures the intended meaning of individual rows as described in Section 3.1.

The semantics of the whole table is given by the two completions of the collections of *Initiates* and *Terminates* rules. These completions (standard in the EC) reflect the implicit information in a given SCR table that combinations of condition values not explicitly identified are not mode transitions. As discussed in Section 3.1 we may regard SCR tables as also containing "hidden" rows (which the engineer does not list) in which the current and the new mode are identical. Violations of system invariants are just as likely to be caused by these "hidden" rows as by the real rows of the table. Because our translation utilises completions, the abductive tool is able to identify problems in "hidden" as well as real rows.

Our EC translation supplies a semantics to mode transition tables that is independent from other parts of the SCR specification. In particular, the translation does not include information about the initial state, and the abductive

tool does not rely on such information to check system invariants. Our technique is therefore also applicable to systems where complete information about the initial configuration of the environment is not available. The abductive tool does not need to use defaults to "fill in" missing initial values for conditions. (Information about the initial state may also be represented; e.g, $HoldsAt(Off, 0)$, so that system invariants may be checked w.r.t. to the initial state separately).

The Abductive Procedure For the purposes of discussion, let us suppose Table 1 has been translated into an EC specification $EC_A(\mathbb{N})$. The system invariants in this particular case are translated into 4 universally quantified sentences $\forall t.I_1(t), \ldots, \forall t.I_4(t)$. In general there will be n such constraints, but we always add an additional constraint $\forall t.I_0(t)$ which simply states (via an exclusive or) that the system is in exactly one mode at any one time. We use the term $\forall t.I(t)$ to stand for $\forall t.I_1(t), \ldots, \forall t.I_n(t)$. For our case study the invariants are (reading "|" as exclusive or):

$$
\begin{array}{ll}
I_0: & [HoldsAt(Off,t) \mid HoldsAt(Inactive,t) \mid \\
 & \qquad HoldsAt(Cruise,t) \mid HoldsAt(Override,t)] \\
I_1: & HoldsAt(Off,t) \equiv \neg HoldsAt(Ignited,t) \\
I_2: & HoldsAt(Inactive,t) \rightarrow [HoldsAt(Ignited,t) \wedge \\
 & \qquad [\neg HoldsAt(Running,t) \vee \neg HoldsAt(Activate,t)]] \\
I_3: & HoldsAt(Cruise,t) \rightarrow [HoldsAt(Ignited,t) \wedge \\
 & \qquad HoldsAt(Running,t) \wedge \neg HoldsAt(Brake,t)] \\
I_4: & HoldsAt(Override,t) \rightarrow [HoldsAt(Ignited,t) \wedge HoldsAt(Running,t)]
\end{array}
$$

As stated previously, Theorem 1 allows us to use our tool with a reduced version of the EC specification that uses a time structure $\mathbb{S}$ consisting of just two points S_c and S_n with $S_c < S_n$. To recap, our abductive procedure attempts to find system behaviors that are counterexamples of the system invariants by generating a consistent set Δ of *HoldsAt* and *Happens* facts (positive or negative literals grounded at S_c), such that $EC(\mathbb{S}) \cup I(S_c) \cup \Delta \models \neg I(S_n)$. We can also check the specification against a particular invariant $\forall t.I_i(t)$ by attempting to abduce a Δ such that $EC(\mathbb{S}) \cup I(S_c) \cup \Delta \models \neg I_i(S_n)$. Because the abductive procedure is complete, failure to find such a Δ ensures that the table satisfies the invariant(s). If, on the other hand, the tool generates a Δ, this Δ is effectively a pointer to a particular row in the table that is problematic. For example, when checking the table against I_3 the tool produces the following:

$$
\begin{array}{rl}
\Delta = & \{HoldsAt(Ignited, S_c), HoldsAt(Running, S_c), HoldsAt(Toofast, S_c), \\
 & \neg HoldsAt(Brake, S_c), HoldsAt(Cruise, S_c), \neg Happens(@F(Ignited), S_c), \\
 & \neg Happens(@F(Running), S_c), \neg Happens(@F(Toofast), S_c), \\
 & Happens(@T(Brake), S_c)\}
\end{array}
$$

Clearly, this Δ identifies one of the "hidden" rows of the table in which a "@T(Brake)" event merely results in the system staying in mode "Cruise". The requirements engineer now has a choice: (1) alter the new mode in this (hid-

den) row so that invariant I_3 is satisfied (in this case the obvious choice is to change the new mode from "Cruise" to "Override", and make this previously hidden row explicit in the table), (2) weaken or delete the system invariant (in this case I_3) that has been violated, or (3) add an extra invariant that forbids the combination of *HoldsAt* literals in Δ (e.g. add $I_5 = [HoldsAt(Cruise, t) \rightarrow \neg HoldsAt(Toofast, t)]$). This example illustrates all the types of choices for change that will be available when violation of an invariant is detected. Choices such as these will be highly domain-specific and therefore appropriate for the engineer, rather than the tool, to select. After the selected change has been implemented, the tool should be run again, and this process repeated until no more inconsistencies are identified.

4 Conclusions, Related and Future Work

Our case study illustrates the two characteristics of our approach mentioned in the introduction. It was able to detect violations of invariants even though the SCR specification used did not include information about an initial state. The counterexamples generated acted as pointers to rows in the mode transition tables and to individual invariants that were problematic. It avoids high computational overheads because of the choice of logical representation and theoretical results, which allow us to reduce the reasoning task before applying the tool. We believe our approach could be more widely applicable. In particular, we are investigating its use in analysing LTS [18] specifications.

A variety of techniques have been developed for analysing requirements specifications. These range from structured inspections [8], to more formal techniques such as model checking, theorem proving [6] and other logic-based approaches (e.g. [20,27,28]). Most techniques based on model checking facilitate automated analysis of requirements specifications and generation of counterexamples when errors are detected [2,4,11]. However, in contrast to our approach they presuppose complete descriptions of the initial state(s) of the system to compute successor states. Moreover, they need to apply abstraction techniques to reduce the size of the state space, and can only handle finite state systems.

For example, in the context of SCR, [11] illustrates how both explicit state model checkers, such as Spin [12], and symbolic model checkers, like SMV [19], can be used to detect safety violations in SCR specifications. The first type of model checking verifies system invariants by means of state exploration. Problems related to state explosion are dealt with by the use of sound and complete abstraction techniques, which reduce the number of variables to just those that are relevant to the invariant to be tested [11]. The goal-driven nature of our abductive EC has the same effect, in that abduction focuses reasoning on goals relevant to the invariant, and the EC ensures that this reasoning is at the level of relevant variables (fluents) rather than via the manipulation of entire states. The essential differences between our approach and this type of model checking are that our system (i) deals with specifications in which information about the initial state is incomplete, and (ii) reports problems in terms of individual

mode transitions (which correspond directly to rows in the tables) rather than in terms of particular paths through a state space. The approach will in certain cases be over-zealous in its reporting of potential errors, in that it will also report problems associated with system states that are in reality unreachable from any possible initial state if such information is given elsewhere in the specifications. However, this feature can only result in overly robust, rather then incorrect, specifications. If desired we can reapply the abductive procedure, with information about the initial state and a full time structure, to test for reachability.

Theorem proving [24] provides an alternative way of analysing requirements specifications, even for infinite state systems. However, in contrast to our approach this does not provide useful diagnostic information when a verification fails, and computations may not always terminate. [5] uses a hybrid approach based on a combination of specialised decision procedures and model checking for overcoming some of the limitations described above. This approach makes use of induction to prove the safety-critical properties in SCR specifications, and so again states identified as counterexamples may not be reachable.

Of logic-based approaches, the work in [28] is particularly relevant. This describes a goal-driven approach to requirement engineering in which "obstacles" are parts of a specification that lead to a negated goal. This approach is comparable to ours in that its notion of goals is similar to our notion of invariants, and its notion of obstacles is analogous to our notion of abducibles. However, the underlying *goal-regression* technique is not completely analogous to our abductive decision procedure. Although it uses backward reasoning and classical unification as in the abductive phase of our decision procedure, no checking for consistency or satisfaction of domain-dependent constraints is performed once an obstacle is generated. Moreover, the identification of obstacles is not automated. Our procedure might also be used effectively to support automated identification of obstacles in [28]'s framework.

Recent work has also demonstrated the applicability of abductive reasoning to software engineering in general. Menzies has proposed the use of abductive techniques for knowledge-based software engineering, providing an inference procedure for "knowledge-level modeling" that can support prediction, explanation, and planning [20]. Satoh has also proposed the use of abduction for handling the evolution of (requirements) specification, by showing that minimal revised specifications can efficiently be computed using logic programming abductive decision procedures [27].

Our next aim is to test our approach on larger and more complex specifications, for example of systems with infinite states or including non-determinism. As mentioned in Section 2.2, the EC allows the representation of such types of specifications, but further experimentation is needed.

Acknowledgements

Thanks to R. Bharadwaj, G. Cugola, P. Grimm, C. Heitmeyer, A. Kakas, B. Labaw, A. van Lamsweerde, J. Moffet, D. Zowghi and the DSE group at

IC. This work was supported by the EPSRC projects MISE (GR/L 55964) and VOICI (GR/M 38582).

References

1. Alspaugh, T. et al. (1988). *Software Requirements for the A-7E Aircraft.* Naval Research Laboratory. 29
2. Anderson, R., et al. (1996). *Model Checking Large Software Specifications.* ACM Proc. of 4th Int. Symp. on the Foundation of Software Engineering. 34
3. Atlee, J. M., and Gannon, J. (1993). *State-Based Model Checking of Event-Driven System Requirements.* IEEE Transaction on Software Engineering, 19(1): 24-40. 29
4. Bharadwaj, R., and Heitmeyer, C. (1997). *Model Checking Complete Requirements Specifications Using Abstraction.* Technical Report No. NRL-7999, NRL. 29, 34
5. Bharadwaj, R., and Sims, S. (2000). *Salsa: Combining Solvers with BDDs for Automated Invariant Checking.* Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in CS, Springer. 35
6. Clarke, M., and Wing, M. (1996). *Formal Methods, State of the Art and Future Directions.* ACM Computing Survey, 28(4): 626-643. 23, 34
7. Easterbrook, S., and Callahan, J. (1997). *Formal Methods for Verification and Validation of Partial Specifications.* Journal of Systems and Software. 29
8. Gilb, T., and Graham, D. (1993). *Software Inspection.* Addison-Wesley. 34
9. Heitmeyer, C. L., Labaw, B., and Kiskis, D. (1995). *Consistency Checking of SCR-style Requirements Specifications.* Proc. of 2nd Int. Symp. on Requirements Engineering, York, 27-29. 29
10. Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G. (1996). *Automated Consistency Checking of Requirements Specifications.* ACM Transaction of Software Engineering and Methodology, 5(3): 231-261. 29
11. Heitmeyer, C. L., et al. (1998). *Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications.* IEEE Transaction on Software Engineering, 24(11): 927-947. 30, 34
12. Holzmann, G. J. (1997). *The Model Checker SPIN.* IEEE Transaction on Software Engineering, 23(5): 279-295. 34
13. Kakas, A. C., and Mancarella, P. (1990). *Generalised Stable Models: A Semantics for Abduction.* ECAI'90, Stockholm, pages 385-391. 28
14. Kakas, A. C., and Michael, A. (1995). *Integrating Abductive and Constraint Logic Programming.* Proc. of 12th Int. Conf. on Logic Programming, Tokyo. 25, 28
15. Kakas, A. C., and Miller R. (1997). *A Simple Declarative Language for Describing Narratives with Actions.* Journal of Logic Programming, Special issue on Reasoning about Actions and Events, 31(1-3): 157-200. 28
16. Kakas, A. C., Kowalski, R. A., and Toni, F. (1998). *The Role of Abduction in Logic Programming.* In C. J. Hogger, J. A. Robinson D. M. Gabbay (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming (235-324). OUP. 23, 24
17. Kowalski, R. A., and Sergot, M. J. (1986). *A Logic-Based Calculus of Events.* New Generation Computing, 4: 67-95. 23
18. Magee, J., and Kramer, J. (1999). *Concurrency: State Models and Java Programs.* John Wiley. 25, 34
19. McMillian, K. L. (1993). *Symbolic Model Checking.* Kluwer Academic. 34

20. Menzies, T. (1996). *Applications of Abduction: Knowledge Level Modeling.* International Journal of Human Computer Studies. 23, 34, 35
21. Miller, R. (1997) *Deductive and Abductive Planning in the Event Calculus.* Poc. 2nd AISB Workshop on Practical Reasoning and Rationality, Manchester, U. K. 24
22. Miller, R., and Shanahan, M. (1999). *The Event Calculus in Classical Logic.* Linkoping Electronic Articles in Computer and Information Science, 4(16). 23, 25, 27
23. Miller, S. (1998). *Specifying the mode logic of a Flitght Guidance System in CoRE and SCR.* Proceedings of 2nd Workshop of Formal Methods in Software Practice. 29
24. Owre, S., et al. (1995). *Formal verification for fault-tolerant architecture: Prolegomena to the design of PVS.* IEEE Transactions on S.E, 21(2): 107-125. 35
25. Parnas, D. L., and Madey, J. (1995). *Functional Documentation for Computer Systems.* Technical Report No. CRL 309, McMaster University. 29
26. Russo, A., Miller, R., Nuseibeh, B., and Kramer, J. (2001). *An Abductive Approach for Analysing Event-based Specifications.* Technical Report no. 2001/7, Imperial College. 25, 27, 28
27. Satoh, K. (1998). *Computing Minimal Revised Logical Specification by Abduction.* Proc. of Int. Workshop on the Principles of Software Evolution, 177-182. 23, 34, 35
28. van Lamsweerde, A., Darimont, R., and Letier, E. (1998). *Managing Conflicts in Goal-Driven Requirement Engineering.* IEEE Transactions on S. E. 34, 35

Trailing Analysis for HAL

Tom Schrijvers[1], Maria García de la Banda[2], and Bart Demoen[1]

[1] Department of Computer Science
K.U.Leuven, Belgium
[2] Department of Computer Science, Monash University, Melbourne

Abstract. The HAL language includes a Herbrand constraint solver which uses Taylor's PARMA scheme rather than the standard WAM representation. This allows HAL to generate more efficient Mercury code. Unfortunately, PARMA's variable representation requires value trailing with a trail stack consumption about twice as large as for the WAM. We present a trailing analysis aimed at determining which Herbrand variables do not need to be trailed. The accuracy of the analysis comes from HAL's semi-optional determinism and mode declarations. The analysis has been partially integrated in the HAL compiler and benchmark programs show good speed-up.

1 Introduction

Mercury [SHC95] is a logic programming language considerably faster than traditional Prolog implementations. One reason is that Mercury requires the programmer to provide type, mode and determinism declarations whose information is used to generate efficient target code. Another reason is that variables can only be ground (i.e., bound to a ground term) or free (i.e., first time seen by the compiler and thus unbound and unaliased). Since neither aliased variables nor partially instantiated structures are allowed, Mercury does not need to support full unification; only assignment, construction, deconstruction and equality testing for ground terms are required. Furthermore, it does not need to perform trailing. This is because trailing aims at storing enough information to be able to reconstruct the previous state upon backtracking. This usually means recording the state of unbound variables right before they become aliased or bound. Since free variables have no runtime representation they do not need to be trailed.

HAL [DdlBH+99b, DdlBH+99a] is a constraint logic language designed to support the construction, extension and use of constraint solvers. HAL also requires type, mode and determinism declarations and compiles to Mercury so as to leverage from its sophisticated compilation techniques. However, unlike Mercury, HAL includes a Herbrand constraint solver which provides full unification. This solver uses Taylor's PARMA scheme [Tay91, Tay96] rather than the standard WAM representation [AK91], because, unlike the WAM, the PARMA representation for ground terms is equivalent to that of Mercury. Thus, calls to the Herbrand constraint solver can be replaced by calls to Mercury's efficient routines whenever ground terms are being manipulated.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 38–53, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Unfortunately, the increased expressive power of full unification comes at a cost, which includes the need to perform trailing. Furthermore, trailing is more expensive in the PARMA scheme than in the WAM. This overhead can however be reduced by performing a trailing analysis that detects and eliminates unnecessary trailings. For traditional logic languages such analysis is rather inaccurate, since little is known about the way predicates are used. For HAL however, determinism information significantly improves accuracy, thus countering the negative aspects of the PARMA scheme and helping retain Mercury-like efficiency.

Next section reviews Taylor's scheme, the trailing of PARMA variables and when it can be avoided. Section 3 summarises the information used by our domain to improve its accuracy. Section 4 presents the `notrail` analysis domain. Section 5 shows how to analyse HAL's body constructs. Section 6 shows how to use the analysis information to avoid trailing. The results of the analysis are summarised in Section 7. Finally, future work is discussed in Section 8.

2 The PARMA Scheme and Trailing

An unbound variable is represented in the PARMA scheme by what is known as a PARMA chain. If the variable is not aliased the chain has length one (a self-reference). For a set of aliased variables the chain is a circularly linked list. Unifying two variables in this scheme consists of cutting their PARMA chains and combining them into one big chain. When a variable becomes bound, all cells in its chain are set to reference the structure that it is bound to. This is unlike the WAM scheme, where only one cell (the one obtained by dereferencing) is set to reference the structure. Hence, checking whether a variable is bound in the PARMA scheme requires no dereferencing, thus increasing efficiency.

As mentioned before, trailing aims at storing enough information regarding the representation state of a variable before each choice-point to be able to reconstruct such state upon backtracking. In the case of PARMA chains the change of representation state occurs at the cell level: from being a self-reference (when the variable pointing to the cell – *the associated variable* – is unbound and unaliased), to pointing to another cell in the chain (when the associated variable gets aliased), to pointing to the final structure (when any variable associated to a cell in the same chain gets bound). Thus what we need to trail are the cells. This is done in HAL using the following code:

```
trail(p, tr)
{
   *(tr++) = *p;
   *(tr++) = p;
}
```

which takes a pointer `p` to a cell in a PARMA chain and the pointer `tr` to the top of the trail. It first stores the contents of the cell and then its address.

Let us now discuss when cells need to be trailed and when this can be avoided. We have seen before that trailing is only needed when the representation state of

a variable changes, and that this can only happen when the variable is unbound and, due to a unification, it becomes either aliased or bound.

Trailing during variable–variable unification Let us start by discussing the information needed to reconstruct the state before aliasing two unbound variables belonging to separate chains. The result of the aliasing is the merging of the two separate chains into a single one. This can be done by changing the state of only two cells: those associated to each of the variables. Since each associated cell appears in a different chain, the final chain can be formed by simply interchanging their respective successors. To be able to reconstruct the previous situation, one just needs to know which two cells have been changed and what their initial value was. This is achieved by the following (simplified) code:

```
trail(X,tr);
trail(Y,tr);
oldX = *X;
X = *Y;
Y = oldX;
```

Notice that `X` and `Y` are trailed independently. As only their associated cells need to be trailed, we will refer to this kind of trailing as *shallow* trailing.

Trailing during variable–nonvariable unification When an unbound variable is bound every single cell in its chain is set to point to the nonvariable term. Thus, we can only reconstruct the chain if *all* cells in the chain are trailed. Note that this is much more expensive than in the WAM scheme where only the dereferenced cell is set to point to the nonvariable term and, therefore, only this cell needs to be trailed. The combined unification-trailing code is as follows:

```
start = X;
do {
   next = *X;
   trail(X,tr);
   *X = T;
   X = next;
} while (X != start);
```

Since all cells in the chain of the unbound variable are trailed, we call this *deep* trailing of the variable.

Unnecessary trailing: There are at least two cases in which the trailing of an unbound variable can be avoided:

- The variable had no representation before the unification (e.g., it was free in Mercury): there is no previous value to remember, so trailing is not required.
- The cells that need to be trailed (the associated cell in the case of variable–variable, all cells in the case of variable–nonvariable) have already been trailed *since the most recent choice-point*. Upon backtracking only the earliest trailing after the choice-point is important, since that is the one which enables the reconstruction of the state of the variable before the choice-point.

3 Language Requirements

The analysis presented in this paper was designed for the HAL language. However, it can be useful for any language that uses PARMA representation and that provides accurate information regarding the following properties:

- Instantiation state: trailing analysis can gain accuracy by taking into account the instantiation state, i.e. whether the variable is new, ground or old. State new corresponds to variables with no internal representation (equivalent to Mercury's free instantiation). State ground corresponds to variables known to be bound to ground terms. In any other case the state is old, corresponding to variables which might be unbound but do have a representation (a chain of length one or more) or bound to a term not known to be ground. Variables with instantiation state new, ground or old will be called new, ground or old variables, respectively. Note that once a new variable becomes old or ground, it can never become new again. And once it is known to be ground, it can safely remain ground. Thus, the three states can be considered mutually exclusive. The information should be available at each program point p as a table associating with each variable in scope of p its instantiation state.
- Determinism: trailing analysis can also gain accuracy from the knowledge that particular predicates have at most one solution. This information should be available as a table associating with each predicate (procedure to be more precise) its inferred determinism.
- Sharing: our trailing analysis can exploit sharing information to increase accuracy. This information should be available at each program point p as a table associating with each variable in scope of p the set of variables which possibly share with it.

4 The `notrail` Analysis Domain

The aim of the `notrail` domain is to keep enough information to be able to decide whether the program variables in a unification need to be trailed or not, so that if possible, optimised versions which do not perform the trailing can be used instead. In order to do this, we must remember that only variables which are unbound at run-time need to be trailed. This suggests making use of the instantiation information mentioned in the previous section.

Let Var_p denote the set of all program variables in scope at p. A lookup in the instantiation table will be represented by the function $inst_p : Var_p \rightarrow \{new, ground, old\}$. This function allows us to partition Var_p into three disjoint sets: New_p, $Ground_p$ and Old_p containing the set of new, ground and old variables, respectively. Assuming that Var_p contains n variables and the tree we have used to implement the underlying table is sufficiently balanced, then, the size of the Old_p is $\mathcal{O}(n)$ and the complexity of $inst_p$ is $\mathcal{O}(\log n)$.

We have already established that variables in New_p and $Ground_p$ do not need to be trailed. Thus, only variables in Old_p need to be represented in the `notrail`

domain. Recall that Old_p not only contains all unbound program variables, but also those bound to terms which the analysis cannot ensure to be ground. This is necessary to ensure correctness: even though variables which are bound do not need to be trailed, program variables might be bound to terms containing one or more unbound variables. It is the trailing state of these unbound run-time variables that is represented through the domain representation of the bound program variable.

Now that we have decided which program variables need to be represented by our domain, we have to decide how to represent them. We saw before that it is unnecessary to trail a variable in a variable–variable unification if its associated cell has already been trailed, i.e., if the variable has already been shallow trailed since the most recent choice-point. For the case of variable–nonvariable unification this is not enough, we need to ensure all cells in the chain have already been trailed, i.e, the variable has already been deep trailed. This suggests a domain which distinguishes between shallow and deep trailed variables. This can be easily done by partitioning Old_p into three disjoint sets of variables with a different trailing state: those which might not have been trailed yet, those which have at least been shallow trailed, and those which have been deep trailed. It is sufficient to represent only two sets to be able to reconstruct the third. Hence, the type of the elements of our `notrail` domain $L_{\texttt{notrail}}$ will be $\mathcal{P}(Old_p) \times \mathcal{P}(Old_p)$, where the first component represents the set of variables which have already been shallow trailed, and the second component represents the set of already deep trailed variables. In the following we will use $l_1, l_2, \ldots$ to denote elements of $L_{\texttt{notrail}}$ at program points $1, 2, \ldots$, and $s_1, s_2, \ldots$ and $dp_1, dp_2, \ldots$ for the already shallow and deep trailed components of the corresponding elements. Also, the elements of the domain will be referred to as descriptions. The descriptions before and after a goal will be referred to as the pre- and post-descriptions, respectively.

Note that, by definition, we can state that if a variable has already been deep trailed, then it has also been shallow trailed (i.e., if all cells in the chain have already been trailed, then the cell associated to the variable has also been trailed). The partial ordering relation $\sqsubseteq$ on $L_{\texttt{notrail}}$ is thus defined as follows:

$$\forall (s_p^1, dp_p^1), (s_p^2, dp_p^2) \in L_{notrail} : (s_p^1, dp_p^1) \sqsubseteq (s_p^2, dp_p^2) \Leftrightarrow \begin{cases} s_p^2 \subseteq dp_p^1 \cup s_p^1 \\ dp_p^2 \subseteq dp_p^1 \end{cases}$$

This implies that deep trailing is stronger information than shallow trailing, and shallow trailing is stronger than no trailing at all. Also note that descriptions are compared at the same program point only (so that the instantiation and sharing information is identical). An example of a trailing lattice is shown in Fig. 1. Clearly $(L_{notrail}, \sqsubseteq)$ is a complete lattice with top description $\top_p = (\emptyset, \emptyset)$ and bottom description $\bot_p = (\emptyset, Old_p)$.

There are two important points that need to be taken into account when considering the above domain. The first point is that the dp_p component of a description will be used not only to represent already deep trailed variables but any variable in Old_p which for whatever reason does not need to be trailed. The reader might then wonder why variables in New_p or $Ground_p$ are not included

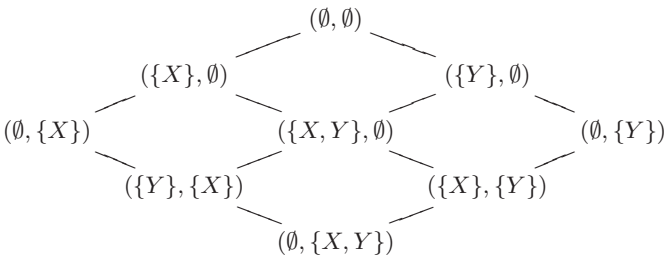


Fig. 1. `Notrail` lattice example

in dp_p since they do not need to be trailed either. This is of course possible. However, this would make most abstract operations slightly more complex.

The second point is that as soon as a deep trailed variable X is known to share with a shallow trailed variable Y, X also must become shallow trailed since some cell in some newly merged chain might come from Y and thus might not have been trailed. We will thus use the sharing information provided at each program point p to define the function $share_p : Old_p \rightarrow \mathcal{P}(Old_p)$, which assigns to each variable in Old_p a set of variables in Old_p that possibly share with it. This information will be used to define the following function which makes trailing information consistent with its associated sharing information:

$$consist_p((s, dp)) = (s \cup x, dp \setminus x)$$

where

$$x = \{X \in dp | (share_p(X) \setminus dp) \neq \emptyset\}$$

From now on we will assume that $\forall (s, dp) \in L_{\texttt{notrail}} : consist_p((s, dp)) = (s, dp)$ and use the *consist* function to preserve this property.

Given HAL's implementation of the sharing analysis domain `ASub` [Søn86] the time complexity of $share_p$ is $\mathcal{O}(n^2)$. Furthermore, since `ASub` explicitly carries the set of ground variables at each program point (g_p), we will use this set rather than computing a new one ($Ground_p$) from the instantiation information, thus increasing efficiency. The major cost of $consist_p$ is the computation of x: for each of the $\mathcal{O}(n)$ variables the $share_p$ set has to be computed. All other set operations are negligible in comparison. Hence, the overall time complexity is $\mathcal{O}(n^3)$. We will see that the complexity of this function determines the complexity of all the operations that use it. Thus, we will use it only when strictly necessary.

In summary, each element $l_p = (s_p, dp_p)$ in our domain can be interpreted as follows. Consider the program variable X. If $X \in dp_p$, this means that all cells in all chains represented by X have already been trailed (if needed). Therefore, X does not need to be trailed in any unification for which l_p is a pre-condition. If $X \in s_p$ we have two possibilities. If X is known to be unbound, then its associated cell has been shallow trailed. Therefore, it does not need to be trailed in any variable-variable unification for which l_p is a pre-condition. If X might be bound, then a cell of one of its chains might not be trailed. As a result, no optimisation can be performed in this case.

We could, of course, represent bound variables more accurately, by requiring the domain to keep track of the different chains contained in the structures to which the program variables are bound, their individual trailing state and how these are affected by the different program constructs. Known techniques (see for instance [JB93, HCC95, MWB94]) based on type information could be used to keep track of the constructor that a variable is bound to and of the trailing state of the different arguments, thereby making this approach possible.

5 Analysing HAL Body Constructs

This section defines the `notrail` operations required by HAL's analysis framework [Net01] to analyse the different body constructs.

Variable–variable unification: $X = Y$. There are several cases to consider:

- If one of the variables (say X) is new, it will simply be assigned a copy of the tagged pointer of Y. No new PARMA chain is created, and thus no trailing is required. The trailing state of X becomes the same as that of Y.
- If one of the variables is ground, the other one will be ground after the unification. Hence, neither of them will appear in the post-description.
- If both variables are deep trailed, the unification only needs to merge the chains (no need to trail again). Hence, both variables remain deep trailed.
- Otherwise, at least one of the variables is not deep trailed. If both variables are unbound, unification will merge both chains while at the same time performing shallow trailing if necessary. Thus after the unification both variables will be shallow trailed. If at least one variable is bound, the other one will become bound after the unification. As stated earlier, bound variables can be treated in the same way.
  Note that if either variable was deep trailed before the unification, all shared variables must become shallow trailed as well after the unification. This requires applying the *consist* function.

Formally, let $l_1 = (s_1, dp_1)$ be the pre-description and g_p be the set of ground variables at program point p. Its post-description l_2 can be obtained as:

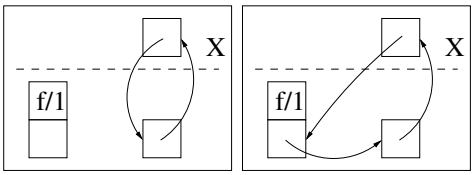
$$l_2 = unify(X, Y) = \begin{cases} same(X, Y, l_1) & , X \text{ is new} \\ remove_ground(l_1, g_2) & , X \text{ is ground} \\ min(X, Y, l_1) & , X \text{ and } Y \text{ are old} \\ unify(Y, X) & , otherwise \end{cases}$$

with

$$remove_ground(l_i, v_i) = (s_i \setminus v_i, dp_i \setminus v_i)$$

$$same(X, Y, (s_1, dp_1)) = \begin{cases} (s_1 \cup \{X\}, dp_1) & , Y \in s_1 \\ (s_1, dp_1 \cup \{X\}) & , Y \in dp_1 \\ (s_1, dp_1) & , otherwise \end{cases}$$

$$min(X, Y, (s_1, dp_1)) = \begin{cases} (s_1, dp_1) & , \{X, Y\} \subseteq dp_1 \\ consist_2((s_1 \cup \{X, Y\}, dp_1 \setminus \{X, Y\})) & , otherwise \end{cases}$$



(a) Before (b) After

Fig. 2. Term construction example: $f(X)$. The dashed line represents a choice-point

Here $remove_ground(l_i, v_i)$ removes all variables in v_i from l_i, $same(X, Y, l_i)$ gives X the same trailing state as Y and $min(X, Y, l_i)$ ensures that X and Y have a shallow trailed state, unless both are deep trailed.

Note that the first case in the definition of *same* does not require a call to *consist* even though dp_1 is modified by adding X to it. This is because X was previously a new variable and, thus, it cannot introduce any sharing.

The worst case time complexity, $\mathcal{O}(n^3)$, is again due to *consist*.

Variable–term unification: $Y = f(X_1, \ldots, X_n)$. There are two cases to consider: If Y is new, the unification simply constructs the term in Y. Otherwise, the term is constructed in a fresh new variable Y' and the unification $Y' = Y$ is executed next. Since unifications of the form $Y' = Y$ have been discussed above, here we only focus on the construction into a new variable.

When a term, e.g. $f(X)$, is constructed with X being represented by a PARMA chain, the argument cell in the structure representation of $f/1$ is inserted in the chain of X (see Fig. 2). While X requires shallow trailing, the cell of the term requires no trailing at all as it is newly created.

Generalised, as the arguments of a new term will share after term construction, they are only allowed to remain deep trailed if all are deep trailed. Otherwise all arguments become shallow trailed. Similarly, Y becomes deep trailed if all arguments are deep trailed, and shallow trailed otherwise.

Formally, let $l_1 = (s_1, dp_1)$ be the pre-description of the unification and x be the set of variables $\{X_1, \ldots, X_n\}$. Its post-description l_2 can be obtained as:

$$l_2 = \begin{cases} (s_1, dp_1 \cup \{Y\}) & , x \subseteq dp_1 \\ (s_1 \cup x \cup \{Y\}, dp_1) & , x \cap dp_1 = \emptyset \\ consist_2((s_1 \cup x \cup \{Y\}, dp_1 \setminus x)) & , otherwise \end{cases}$$

The worst case time complexity is $\mathcal{O}(n^3)$. This definition can be combined with the previous one for the overall definition of variable–term unification. The implementation can be more efficient, but the complexity will still be $\mathcal{O}(n^3)$.

Predicate call: $p(X_1 \ldots X_n)$. Let l_1 be the pre-description of the predicate call and x the set of variables $\{X_1, \ldots, X_n\}$. The first step will be to project l_1 onto x

resulting in description l_{proj}. Note that onto-projection is trivially defined as:

$$onto_proj(l, v) = (s \cap v, dp \cap v)$$

The second step consists in extending l_{proj} onto the set of variables local to the predicate call. Since these variables are known to be new (and thus they do not appear in Old_1), the extension operation in our domain is trivially defined as the identity. Thus, from now on we will simply disregard the extension steps required by HAL's framework.

The next step depends on whether the predicate is defined by the current module or by another (imported) module. Let us assume the predicate is defined by the current module and let l_{answer} be the answer description resulting from analysing the predicate's definition for calling description l_{proj}. In order to obtain the post-description, we will make use of the determinism information. Thus, the post-description l_2 can be derived by combining the l_{answer} and l_1, using the determinism of the predicate call as follows:

- If the predicate might have more than one answer, then l_2 is equal to l_{result} except for the fact that we have to apply the *consist* function in order to take into account the changes in sharing. This means that all variables that are not arguments of the call, become not trailed.
- Otherwise, l_2 is the result of combining l_{answer} and l_1: the trailing state of variables in x is taken from l_{answer}, while that of other variables is taken from l_1. Any deep trailed variables that share with non-deep trailed variables must, of course, become shallow trailed.

Formalised, the combination[1] function is defined as:

$$\begin{aligned} l_2 &= comb(l_1, l_{answer}) \\ &= \begin{cases} consist(((s_1 \setminus x) \cup s_{answer}, (dp_1 \setminus x) \cup dp_{answer})) & \text{, at most one solution} \\ consist(l_{answer}) & \text{, otherwise} \end{cases} \end{aligned}$$

Obviously the complexity is $\mathcal{O}(n^3)$ because of *consist*.

Example 1. Assume that the call `p(X)` has pre-description $(\{X, Y\}, \emptyset)$ and the predicate `p/1` has answer description $(\{X\}, \emptyset)$. The post-description of the call depends on the determinism of the predicate. If the predicate has at most one solution, the post-description will be $((\{X, Y\} \setminus \{X\}) \cup \{X\}, (\emptyset \setminus \{X\}) \cup \emptyset) = (\{X, Y\}, \emptyset)$. Otherwise the post-description will be equal to the answer description, $(\{X\}, \emptyset)$.

Now, if the predicate is defined in an imported module, we will use the analysis registry created by HAL for every exported predicate: a table containing all

[1] Note that the combination is not the meet of the two descriptions. It is the "specialised combination" introduced in [dlBMSS98] which assumes that l_{answer} contains the most accurate information about the variables in x, the role of the combination being just to propagate this information to the rest of variables in the clause.

call-answer description pairs encountered during analysis. Thus, we do a simple look-up in this table and check if the predicate has a call-answer pair with a call description equal to l_{proj}. If not, then we will choose the smallest call-description less precise than l_{proj}. Since the table *always* includes a pair with the most general ($\top$) calling description, this selection process always finds an appropriate variant. Finally, we combine the answer description l_{answer} of this call-answer pair with l_1 in the same way as for the intramodule call. HAL built-ins are treated in a similar way: a table is available containing the call-answer pairs for every built-in predicate.

Disjunction: $(G_1; G_2; \ldots; G_n)$. Disjunction is the reason why trailing becomes necessary. As mentioned before, trailing might be needed for all variables which were already old before the disjunction. Thus, let l_0 be the pre-description of the entire disjunction. Then, $\top$ will be the pre-description of each G_i except for G_n whose pre-description is simply l_0 (since the disjunction implies no backtracking over the last branch).

Let $l_i = (s_i, dp_i), 1 \leq i \leq n$ be the post-description of goal G_i. We will assume that the set v_i of variables local to each G_i has already been projected out from l_i, where out-projection is identical to *remove_ground*, which has time complexity $\mathcal{O}(n)$. The end result l_{n+1} of the disjunction is the least upper bound (lub) of all branches[2], which is defined as:

$$l_1 \sqcup \ldots \sqcup l_n = consist_{n+1}(remove_ground((s, dp), g_{n+1}))$$

where

$$\begin{aligned} s &= (s'_1 \cap \ldots \cap s'_n) \setminus dp \\ dp &= (dp'_1 \cap \ldots \cap dp'_n) \\ s'_i &= s_i \cup dp'_i \\ dp'_i &= dp_i \cup g_i \end{aligned}$$

Intuitively, all variables which are deep trailed in all descriptions are ensured to remain deep trailed; all variables which are trailed in all descriptions but have not always been deep trailed (i.e., are not in dp) are ensured to have already been (at least) shallow trailed. Note that variables which are known to be ground in all descriptions (those in g_{n+1}) are eliminated. This is consistent with the view that only old variables are represented by the descriptions.

Example 2. Let $l_0 = (\emptyset, \{X, Y, Z\})$ be the pre-description of disjunction:

```
( X = Y ; X = f(Y, Z) )
```

Let us assume there is no sharing at that program point. Then, the pre-descriptions of the first unification is $(\emptyset, \emptyset)$, the $\top$ element of our domain. The pre-description of the second unification is $(\emptyset, \{X, Y, Z\})$, i.e., since this is the last branch in the disjunction, its pre-description is identical to the pre-description of the entire disjunction. Their post-descriptions are $(\{X, Y\}, \emptyset)$ and $(\emptyset, \{X, Y, Z\})$, respectively. Finally, the lub of the two post-descriptions results in $(\{X, Y\}, \emptyset)$.

[2] Note that this is not the lub of the `notrail` domain alone, but that of the product domain which includes sharing (and groundness) information.

The time complexity of the joining of the branches is simply that of the lub operator ($\mathcal{O}(n^3)$) for a fixed maximum number of branches, and it is completely dominated by the $consist_{n+1}$ function.

If-then-else: $I \rightarrow T ; E$. Although the if-then-else could be treated as $(I, T; E)$, this is rather inaccurate since only one branch will ever be executed and, thus, there is no backtracking between the two branches.

Let l_1 be the pre-description to the if-then-else. Then l_1 will also be the pre-description to both I and E. Let l_I be the post-description obtained for I. Then l_I will also be the pre-description of T. Finally, let l_T and l_E be the post-descriptions obtained for T and E, respectively. Then, the post-description for the if-then-else can be obtained as the lub $l_T \sqcup l_E$.

Note that this operation assumes a Mercury-like semantics of the if-then-else: No variable that exists before the if-then-else should be bound or aliased in such a way that trailing is required for backtracking if the condition fails. This is not a harsh restriction, since it is ensured whenever the if-condition is used in a logical way, i.e., it simply inspects existing variables and does not change any non-local variable. The time complexity of the joining of the branches is again $\mathcal{O}(n^3)$, just like the operation over the disjunction.

Example 3. Let $l_0 = (\emptyset, \emptyset)$ be the pre-description of the if-then-else:

```
( N = 1 →
        X = Y
;
        X = f(Y, Z)
)
```

Assume no variables share before the if-then-else. Then, l_0 is equal to the pre-description of both the then- and else-branch. The post-description of the then-branch is $(\{X, Y\}, \emptyset)$ and that of the else-branch is $(\{X, Y, Z\}, \emptyset)$. The post-description finally is obtained by taking their lub: $(\{X, Y\}, \emptyset)$.

Higher-order unification: $Y = p(X_1, \ldots, X_n)$. This involves the creation of a partially evaluated predicate, i.e., we are assuming there is a predicate with name p and arity equal or higher than n for which the higher-order construct Y is being created. In HAL, Y is required to be new. Also, it is often too difficult or even impossible to know whether Y will be called or not and, if so, where. Thus, HAL follows a conservative approach and requires that the instantiation of the "captured" arguments (i.e., $X_1, \ldots, X_n$) remain unchanged after executing the predicate.

The above requirements allow us to follow a simple (although conservative) approach: Only after a call to Y will the trailing of the captured variables be affected. If the predicate might have more than one solution and thus may involve backtracking, then the involved variables will be treated safely in the analysis at the call location if they are still statically live there.

If the predicate does not involve backtracking, then trailing information might not be inferred correctly at the call location if the call contains unifications. This is because the captured variables are generally not known at the

call location. To keep the trailing information safe any potential unifications have to be accounted for in the higher-order unification. Since the predicate involves no backtracking and all unifications leave the variables they involve at least shallow trailed, it is sufficient to demote all captured deep trailed variables to shallow trailed status, together with all sharing deep trailed variables.

Formally, let $l_1 = (s_1, dp_1)$ be the pre-description of the higher-order unification and x be the set of variables $\{X_1, \ldots, X_n\}$. Then its post-description l_2 can be obtained with a time complexity of $\mathcal{O}(n^3)$ as:

$$l_2 = \begin{cases} consist_2((s_1 \cup (x \cap dp_1), dp_1 \setminus x)) & , x \cap dp_1 \not= \emptyset \\ l_1 & , otherwise \end{cases}$$

Notice that between the time of higher-order unification and call, any number of disjunctions could occur. This means that the trailing state of the captured variables at the time of higher-order unification cannot generally be used to select another variant of the predicate than the one with top calling description.

Higher-order call: call$(P, X_1, \ldots, X_n)$. The exact impact of a higher-order call is difficult to determine in general. Fortunately, even if the exact predicate associated to variable P is unknown, the HAL compiler still knows its determinism. This can help us improve accuracy. If the predicate might have more than one solution, all variables must become not trailed. Since the called predicated is typically unknown, no answer description is available to improve accuracy.

Otherwise, the worst that can happen is that the deep trailed arguments of the call become shallow trailed. So in the post-description we move all deep trailed arguments to the set of shallow trailed variables, together with all variables they share with. Recall that for this case the captured variables have already been taken care of at the higher-order unification.

The sequence of steps is much the same as that for the predicate call. First, we project the pre-description l_1 onto the set x of variables $\{X_1, \ldots, X_n\}$, resulting in l_{proj}. Next, the answer description l_{answer} of the higher-order is computed as indicated above:

$$l_{answer} = \begin{cases} (s \cup dp, \emptyset) & , \text{at most one solution} \\ (\emptyset, \emptyset) & , \text{otherwise} \end{cases}$$

Finally, the combination of l_{answer} and l_1 is computed to obtain the post-description l_2.

6 Trailing Optimisation

The optimisation phase consists in deciding for each unification in the body of a clause, which variables need to be trailed. This decision is based on the pre-description of the unification, inferred by the trailing analysis. If some variables do not need to be trailed, the general unification predicate is replaced with an alternative variant that does not trail those particular variables. Thus, we will need a different variant for each possible combination of variables that do and do not need to be trailed.

Table 1. Compilation statistics

Benchmark	Compilation Time			Old unifications	
	Analysis	Total	Percentage	Improved	Total
icomp	34.670	84.760	40.9%	300	1269
hanoi_difflist	.630	7.720	8.2%	13	13
qsort_difflist	.500	32.590	1.5%	7	7
serialize	1.950	8.120	24.0%	10	17
warplan	22.120	98.160	22.5%	69	1392
zebra	2.830	12.550	22.5%	41	178

- For the unification of two unbound variables trailing is omitted for either variable if it is shallow trailed or deep trailed in the pre-description.
- For the binding of an unbound variable trailing is omitted if the variable is deep trailed in the pre-description.
- For the unification of two bound variables the trailing for chains in the structure of either is omitted if the variable is deep trailed in the pre-description.

Often it is not known at compile time whether a variable is bound or not, so a general variable-variable unification predicate is required that performs runtime boundness tests before selecting the appropriate kind of unification. Various optimised variants of this general predicate are needed as well.

Finally, we must point out that term construction with old unbound arguments also involves trailing. This is because each argument is copied into the term structure and if a copy appears to be a pointer to a PARMA chain (the argument was an old unbound variable), then the copy is transformed into a new cell that is added in front of the cell C associated with the argument variable X. Since cell C is modified in the process it requires trailing. However, if X is either shallow or deep trailed, then trailing can be omitted.

7 Results

The analysis has been implemented in the analysis framework of HAL and applied to six HAL benchmarks that use the Herbrand solver: icomp, hanoi_difflist, qsort_difflist, serialize, warplan and zebra. The pre-descriptions inferred for the unifications have then been used to optimise the generated Mercury code by the omission of trailing, as explained in the previous section. Table 1 shows compilation statistics of the benchmarks: compilation time in seconds, measured on an Intel Pentium 166 MHz 96 MB, and the number of improved unifications compared to the total number of unifications involving old variables. The compilation times are high for most benchmarks because most predicates have many call descriptions to consider, something the analysis has not been optimised for yet. For the hanoi_difflist and qsort_difflist benchmarks, the analysis infers that all unifications should be replaced by a non-trailing alternative. In the other benchmarks a much smaller fraction of unifications can be improved due to the heavy use of non-deterministic predicates.

Table 2. Timings and trailings

Benchmark	iterations	Time		Speed-up	Value trailings	
		unoptimised	optimised		unoptimised	optimised
icomp	12,500	1.033	.992	4.0%	146	121
hanoi_difflist	2,500	.973	.739	24.0%	190	0
qsort_difflist	25,000	1.005	.775	22.9%	151	0
serialize	12,500	1.083	1.004	7.3%	212	162
warplan	10	1.663	1.613	3.0%	10,229	10,229
zebra	200	1.124	1.049	6.7%	25,769	24,618

Table 2 presents the timing results of each benchmark for the given number of times it is performed, and compares the number of value trailings of the unoptimised and optimised versions of the benchmarks for a single run. Timing results were obtained on an Intel Pentium 4 1.50 GHz 256 MB.

The huge speed-up of hanoi_difflist and qsort_difflist can be explained by the complete elimination of value trailing. For the other benchmarks the number of eliminated trailings and the speed-up are a lot smaller. This is partly explained by the smaller fraction of improved unification predicates and partly because for some of the improved general unifications the variables are always bound and thus there is no trailing to avoid.

8 Related and Future Work

In [SD02] an improved PARMA trailing scheme is proposed that nearly halves the maximal trail stack for conditional trailing. Also in the HAL-Mercury setting a maximal trail stack reduction of 30% to 50% is obtained. The analysis has been adapted for this improved trailing scheme, and, although it yields less additional improvement, the combination with this trailing scheme is more effective.

A somewhat similar analysis for detecting variables that do not have to be trailed is presented by Debray in [Deb92] together with corresponding optimisations. Debray's analysis however is for the WAM variable representation in a traditional Prolog setting, i.e., without type, mode and determinism declarations. Our analysis could not be easily adapted to the WAM, as it relies heavily on the PARMA variable representation.

Since the presented analysis aims at avoiding trailing, it is worth comparing with runtime methods. If the underlying implementation has an ordered heap (like the WAM) the conditional trailing test is simple: the address of the cell is compared with the heap top of the most recent choice-point. In other cases, the conditional trailing test must rely on some form of time stamping. It is clear that runtime tests avoid **all** unnecessary trailing and analysis cannot improve this. Also, our benchmarks show that there is little to improve (see results for hanoi and qsort). Finally, the runtime tests can be combined with the analysis, but our feeling is that this should not be done with the expensive time stamping mechanism. Note that the technique of [Noy94] for avoiding double trailing of the same cell cannot be applied in the PARMA context, because it relies on only

one cell in a chain (the end for WAM) being subject to trailing. Also note that in the Aquarius system [RD92], when trailing analysis cannot avoid the trailing of a variable, no runtime method for avoiding trailing is used instead.

The proposed analysis achieves very good speed-ups for deterministic benchmarks like hanoi_difflist and qsort_difflist. However, it can be significantly improved through, for example, the use of other kinds of information such as liveness of variables. Also, we would like to investigate the trade-off between analysis complexity and gain. Maybe a slightly less complex analysis would yield a similar speed-up. Alternatively, a more complex analysis that keeps track of the configuration of chains and terms could significantly improve speed.

References

[AK91] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991. 38

[DdlBH+99a] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. Herbrand Constraint Solving in HAL. In *International Conference on Logic Programming*, pages 260–274, 1999. 38

[DdlBH+99b] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. An Overview of HAL. In *Principles and Practice of Constraint Programming*, pages 174–188, 1999. 38

[Deb92] S. Debray. A Simple Code Improvement Scheme for Prolog. *Journal of Logic Programming*, 13(1):349–366, May 1992. 51

[dlBMSS98] M. García de la Banda, K. Marriott, P. Stuckey, and H. Søndergaard. Differential methods in logic program analysis. *JLP*, 35:1–37, 1998. 46

[HCC95] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *JLP*, 22:179–209, 1995. 44

[JB93] G. Janssens and M. Bruynooghe. Deriving descriptions of possible value of program variables by means of abstract interpretation. *JLP*, 13:205–258, 1993. 44

[MWB94] A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure dataflow analysis for prolog. *ACM TOPLAS*, 16:205–258, 1994. 44

[Net01] N. Nethercote. The Analysis Framework of HAL. Master's thesis, University of Melbourne, September 2001. 44

[Noy94] J. Noyé. *Elagage de contexte, retour arrière superficiel, modifications réversibles et autres: une étude approfondie de la WAM.* PhD thesis, Université de Rennes I, November 1994. 51

[RD92] P. Van Roy and A. Despain. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer*, 25(1):54–68, 1992. 52

[SD02] T. Schrijvers and B. Demoen. Combining an Improvement to PARMA Trailing with Analysis in HAL. Technical Report 338, K.U.Leuven, April 2002. 51

[SHC95] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, pages 499–512, February 1995. 38

[Søn86] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986. 43

[Tay91] A. Taylor. *High Performace Prolog Implementation*. PhD thesis, Basser Department of Computer Science, June 1991. 38

[Tay96] A. Taylor. Parma - Bridging the Performance GAP Between Imperative and Logic Programming. *Journal of Logic Programming*, 29(1-3):5–16, 1996. 38

Access Control for Deductive Databases by Logic Programming

Steve Barker

Department of Computer Science, King's College
London, U.K, WC2 2LS

Abstract. We show how logic programs may be used to protect deductive databases from the unauthorized retrieval of positive and negative information, and from unauthorized insert and delete requests. To achieve this protection, a deductive database is expressed in a form that is guaranteed to permit only authorized access requests to be performed. The protection of the positive information that may be retrieved from a database and the information that may be inserted are treated in a uniform way as is the protection of the negative information in the database, and the information that may be deleted.

1 Introduction

Deductive databases are predicted to become increasingly significant in the future [17]. However, in order for deductive databases to make an impact, it is important to consider practical issues like ensuring the security of the information they contain.

Despite the long recognized importance of database security, the security of deductive databases has been largely neglected in the literature. We address this gap by showing how a range of access control policies may be represented by using logic programs, and how these programs may be used to protect a deductive database from unauthorized access requests made by authenticated users.

In the approach we describe, stratified logic programs are employed to represent *role-based access control (RBAC)* policies [20] that specify the authorized retrieval and update operations which users may perform on the objects in a deductive database (i.e., n-place predicates). Our analysis of access policy requirements suggests that most practical RBAC policies may be expressed in stratified logic (and all realistic RBAC policies may be represented as a finite set of locally stratified clauses).

In RBAC, the most fundamental notion is that of a role. A role is defined in terms of a job function in an organization (e.g., a *doctor* role in a medical environment), and users and access privileges on objects are assigned to roles. RBAC policies have a number of well documented attractions, and are widely used in practice [11]. The logic programs we describe are based on the $Hierarchical$ ($level$ $2A$) $RBAC$ model from [20]. Henceforth, we refer to these

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 54–69, 2002.
© Springer-Verlag Berlin Heidelberg 2002

logic programs as $RBAC_{H2A}$ programs. In addition to the assignment of users and access privileges on objects to roles, $RBAC_{H2A}$ programs include a definition of a *role hierarchy* [20].

In [6] and [9], modal logics are considered for specifying confidentiality requirements on deductive databases. However, since these proposals do not consider updates they are limited in scope. Moreover, the languages they employ, and approaches they suggest for protecting deductive databases against unauthorized operations are not especially compatible with the methods of representation and computation that deductive databases typically employ. In contrast, we use stratified logic programs to define the protection of deductive databases precisely because this language enables a declarative specification of an $RBAC_{H2A}$ policy to be formulated and seamlessly incorporated into "mainstream" representations of deductive databases (i.e., $Datalog^{\neg}$ databases).

In overview, our approach is to "compile" access control information into a deductive database. By "compiling" access control information into a deductive database, we mean that the database is written in a form that ensures that only authorized access requests are permitted on it. Hence, our approach is based on *partial evaluation* [13], and involves specializing a deductive database by using access control information.

In other related work, Jajodia *et al.* use clause form logic to specify access control policies for protecting "systems" [14]. However, no computational or implementation details are discussed in [14], and no details are given of applications of the approach. In contrast, our approach is based on the representation of an RBAC policy, involves "compiling" access control information into a deductive database specifically, and describes not only how decisions on the legitimacy of retrieval and update requests are made, but also demonstrates how authorized updates may be computed. In [15], a query language is described which extends *Datalog* and enables a user of a *multilevel secure (MLS) system* [7] to reason about the beliefs held by users with different security clearances. In our approach RBAC rather than MLS policies are specified to protect a deductive database, theorem-proving techniques are used to retrieve the information a user is authorized to see (rather than being used to reason about the beliefs of users), and updates are considered as well as retrievals.

In this paper we extend previous work of ours (see [2]) to show how to formulate deductive databases which enable:

- the negative information as well as the positive information in a deductive database to be specified as being protected from unauthorized retrieval requests.
- a user to distinguish between information which is true or false in a deductive database and information that the user is not permitted to know the truth value of due to access control restrictions.
- the retrieval of negative information and the deletion of information from a deductive database to be treated in a uniform manner.
- the retrieval of positive information and the insertion of information into a deductive database to be treated in a uniform way.

In the form of protected deductive databases described in [2], it is not possible to distinguish between information that is false in the database and information that is false because a user is not authorized to know that the information is true. Moreover, retrievals and updates are not treated in a uniform way, and the protected form of a deductive database requires that non-stratified logic be used if the database includes some clause with a negative condition. In this paper we address these problems by introducing *extended protected databases (EPDs)*.

The rest of the paper is organized thus. In Section 2, some basic notions in logic programming, deductive databases, and RBAC are briefly described. In Section 3, we show how $RBAC_{H2A}$ programs may be represented. In Section 4, we describe the representation of EPDs. In Section 5, some computational issues relating to EPDs are considered. In Section 6, some performance results are discussed. Finally, in Section 7, some conclusions are drawn and suggestions for further work are made.

2 Basic Concepts

A *deductive database* consists of a finite set of clauses, and is usually viewed as comprising two distinct parts:

Definition 1 *The set of ground atomic assertions (i.e., facts) in a deductive database is called the extensional database* (EDB).

Definition 2 *The set of clauses with a non-empty body which appear in a deductive database defines the intensional database* (IDB).

An EPD is a deductive database which ensures that access requests are only possible if an $RBAC_{H2A}$ program that is associated with the EPD specifies that the requested access is authorized. Henceforth, an arbitrary EPD will be denoted by $D*$, and $\Delta D*$ will be used to denote an updated state of $D*$. The EDB and IDB of $D*$ will be denoted by $EDB(D*)$ and $IDB(D*)$, respectively. To denote specific instances of $D*$ ($\Delta D*$), we use D_i* (ΔD_i*) where i is a natural number. We also use $S*$ to denote an arbitrary $RBAC_{H2A}$ program that is used to constrain access to $D*$, and we use S_i* to denote a specific instance of an $RBAC_{H2A}$ program.

An $RBAC_{H2A}$ program is defined on a domain of discourse that includes:

1. A set U of *users*.
2. A set O of *objects*.
3. A set A of *access privileges*.
4. A set R of *roles*.

The U, O, A and R sets, comprise the (disjoint and finite) user, object, access privilege and role identifiers that form part of the universe of discourse for an $RBAC_{H2A}$ program. In this framework we have:

Definition 3 *An authorization is a triple* (u, a, o) *which denotes that a user* u *(*$u \in U$*) has the a access privilege (*$a \in A$*) on the object* o *(*$o \in O$*).*

Definition 4 *If a is an access privilege and o is an object then a permission is a pair (a, o) which denotes that a access is permitted on o.*

Definition 5 *A permission-role assignment is a triple (a, o, r) which denotes that the permission (a, o) is assigned to the role r.*

Definition 6 *A user-role assignment is a pair (u, r) which denotes that the user u is assigned to the role r.*

In this paper, we assume that an $RBAC_{H2A}$ program which defines a *closed policy* [7] with no *session management* [20] is used to protect an EPD.

Definition 7 *In a closed policy, a user u can exercise the a access privilege on object o iff authorization (u, a, o) holds.*

The implementation of other types of access policies for EPDs necessitates that only minor modifications be made to the approach we describe in the sequel. A *security administrator (SA)* is responsible for specifying whatever form of $RBAC_{H2A}$ program an organization requires to protect its databases.

An $RBAC_{H2A}$ role hierarchy is represented by a partial order $(R, \geq)$ that defines a seniority ordering $\geq$ on a set of roles R. Role hierarchies in $RBAC_{H2A}$ are used to represent the idea that, unless constraints are imposed, "senior roles" (more powerful roles) inherit the permissions assigned to "junior" (less powerful) roles in an $RBAC_{H2A}$ role hierarchy. Hence, if $r_i \geq r_j$ then r_i inherits the permissions assigned to r_j.

In the sequel, the constants that appear in clauses will be denoted by symbols that appear in the lower case; variables will be denoted by using upper case symbols. Moreover, we will use u, o, a and r to identify a distinct, arbitrary user, object, access privilege and role, respectively, and we use U, O, A and R to respectively denote an arbitrary set of users, objects, access privileges and roles. It should also be noted that $D* \cup S*$[1] is completely hidden from "ordinary" users to restrict a user's scope for drawing inferences about secret information in an EPD.

Since $D* \cup S*$ is expressed as a stratified program, it follows that the *well-founded semantics* [22] may be used for the declarative semantics for EPDs, and the restricted form of *SLG-resolution* [8] that is sufficient for goal evaluation on stratified programs may be used for the corresponding procedural semantics. When SLG-resolution is used with $D* \cup S*$, a *search forest* is constructed starting from the *SLG-tree* with its root node labeled by the *goal clause* $Q \leftarrow Q$ [8]. From the soundness and (search space) completeness of SLG-resolution (for flounder-free computations), we have:

Proposition 1 $\mathrm{Q} \in \mathrm{WFM}(D* \cup S*)$ *iff there is an SLG-derivation for* $\mathrm{Q} \leftarrow \mathrm{Q}$ *on $D* \cup S*$ that terminates with the answer clause* $\mathrm{Q} \leftarrow$ *(where* $\mathrm{WFM}(D* \cup S*)$ *is the well-founded model of $D* \cup S*$).*

[1] By $D* \cup S*$ we mean the union of the sets of clauses in $D*$ and $S*$.

If an $RBAC_{H2A}$ program is expressed using built-in comparison or mathematical operators then we assume that a SA will express these definitions in a *safe* form [21] such that the arguments of a condition involving these operators are bound prior to the evaluation of the condition.

3 Representing $RBAC_{H2A}$ as a Logic Program

$RBAC_{H2A}$ programs were first described by us in [1]. In [1], a user U is specified as being assigned to a role R by a SA using definitions of a 2-place $ura(U, R)$ predicate. To record that the A access privilege on an object O is assigned to a role R, clause form definitions of a 3-place $pra(A, O, R)$ predicate are used. In the context of deductive databases, the data objects to be protected are n-place EDB and IDB predicates.

In [1], an $RBAC_{H2A}$ role hierarchy $(R, \geq)$ is expressed by a set of clauses that define a *senior_to* relation as the reflexive-transitive closure of an irreflexive-intransitive ds relation that defines the set of pairs of roles (r_i, r_j) such that r_i is directly senior to role r_j, written $ds(r_i, r_j)$, in an $RBAC_{H2A}$ role hierarchy where:

$$\forall r_i, r_j [ds(r_i, r_j) \leftrightarrow r_i \geq r_j \wedge r_i \not= r_j \wedge \neg \exists r_k [r_i \geq r_k \wedge r_k \geq r_j \wedge r_i \not= r_k \wedge r_j \not= r_k]]$$

In clause form logic, the *senior_to* relation may be defined in terms of ds thus (where '_' is an anonymous variable):

$$\begin{aligned} &senior_to(R1, R1) \leftarrow ds(R1, _).\\ &senior_to(R1, R1) \leftarrow ds(_, R1).\\ &senior_to(R1, R2) \leftarrow ds(R1, R2).\\ &senior_to(R1, R2) \leftarrow ds(R1, R3), senior_to(R3, R2). \end{aligned}$$

The *senior_to* predicate is used in the definition of *permitted* that follows:

$$permitted(U, A, O) \leftarrow ura(U, R1), senior_to(R1, R2), pra(A, O, R2).$$

The *permitted* clause expresses that a user U has A access on object O if U is assigned to a role $R1$ that is senior to the role $R2$ in an $RBAC_{H2A}$ role hierarchy defined by the $RBAC_{H2A}$ program, and $R2$ has been assigned A access on O. That is, U has the A access on O if U is assigned to a role that inherits the A access privilege on O.

Since all realistic $RBAC_{H2A}$ programs may be expressed using a set of (locally) stratified clauses, we have:

Theorem 1 *Every $RBAC_{H2A}$ program has a unique perfect model [18].*

Moreover, since the (2-valued) perfect model of a stratified program coincides with the total well-founded model of the program [8], the perfect model of an $RBAC_{H2A}$ program is identical to the 2-valued well-founded model of the

program. An important corollary of $RBAC_{H2A}$ programs being categorical and having a 2-valued model is that $RBAC_{H2A}$ programs define a consistent and unambiguous set of authorizations.

4 EPDs: Representational Issues

For EPDs, we propose the following retrieval semantics:

- if U requests to know whether Q is true in $D*$ then U's query will be successful if U is authorized to know that Q is true in $D*$ and Q is true in U's authorized view of $D*$ (i.e., the subpart of the database the user can retrieve from).
- if U requests to know whether Q is false in $D*$ then U's query will be successful if U is authorized to know that Q is false in $D*$ and Q is false in U's authorized view of $D*$.
- A user's request to know that Q is true (false) in $D*$ will fail if U has insufficient permission to know that Q is true (false) in $D*$ or if Q is false (true) in U's authorized view of $D*$.

To compute the retrieval semantics for EPDs, we use a decision procedure that answers "yes" (with the *answer substitution* Θ [16]) if U requests to know whether Q is true in $D*$, $Q\Theta$ is true in U's authorized view of $D*$, and U is authorized to know that $Q\Theta$ is true in $D*$. Similarly, the procedure will answer "yes" if U requests to know whether Q is false in $D*$, Q is false in U's authorized view of $D*$, and U is authorized to know that Q is false in $D*$. Conversely, the decision procedure will answer "no" if U's request to know that Q is true (false) in $D*$ fails due to Q being false (true) in U's authorized view of $D*$ or because U has insufficient authorization to know that Q is true (false) in $D*$. In the case of a "no" answer to U's request to know whether Q is true (false) in $D*$, U may request to know whether Q is false (true) in $D*$. If U's request to know whether Q is true fails and also U's request to know whether Q is false fails then U can only infer that he or she has insufficient authorization to know the truth value of Q in $D*$.

In EPDs, the retrieval semantics which we adopt are chosen, in part, to enable a candidate update semantics to be defined for EPDs. In EPDs, requests to know what is true in a database are treated in a similar way to insert requests, and requests to know what is false are treated in a similar way to delete requests. That is, we view a request to insert a ground instance of an i-place atom p into an EPD $D*$ as a request to make this instance of p true in $D*$; a positive query involving p is a request to know whether p is true in $D*$. Conversely, we view a request to delete a ground instance of p from $D*$ as a request to make this instance of p false in $D*$ whereas a negative query, of the form *not* p, is a request to know whether p is false in $D*$. By relating update requests to retrieval requests in EPDs, it is possible to use the insert protected form of a predicate p in an EPD as also the read protected form of p in $D*$. Moreover, the delete

protected form of p may be used as the read protected form of $\neg p$ in $D*$ (i.e., the read protection for the negative information in $D*$).

It follows from the discussion above that in an EPD, $D*$, there are two types of protected clause of interest:

1. TYPE 1 clauses: protect the positive information in $D*$ from unauthorized retrieval requests, and protects $D*$ from unauthorized insert requests.
2. TYPE 2 clauses: protect the negative information in $D*$ from unauthorized retrieval requests, and protects $D*$ from unauthorized delete requests.

For updates on $D*$, the change requests that "ordinary" users may make are for the insertion or deletion of a ground atomic instance of an object O that is either explicit in $D*$ (i.e., in $EDB(D*)$) or implicit in $D*$ (i.e., in $IDB(D*)$). The update requests of "ordinary" users may be satisfied only by changing $EDB(D*)$. We believe that only the SA or other trusted users should be permitted to make changes other than the insertion or deletion of ground assertions into or from $D*$.

4.1 Representing TYPE1 IDB Information in EPDs

To protect the clause $I \leftarrow A1, \ldots, Am, not\ B1, \ldots, not\ Bn$, where I is a k-place IDB predicate, in the case of TYPE 1 information, I is expressed in an EPD thus:

$$holds(U, P, +, N, I) \leftarrow permitted(U, P, I), holds(U, P, +, N, A1), \ldots \\ \ldots, holds(U, P, +, N, Am), \\ holds(U, N, -, P, B1), \ldots, holds(U, N, -, P, Bn).$$

In the case of a request to insert a ground instance of I into $D*$, P=*insert* and N=*delete*. In the case where a user's request is to know whether I is true in $D*$, P=*true* and N=*false*.

The '+' symbol in the triple of arguments *(P,+,N)* in *holds* is used to denote that P is a positive operation; N is used to denote the negative counterpart of P. The negative counterpart of P when P=*insert* is *delete* (i.e., N=*delete*); the negative counterpart of P when P=*true* is *false* (i.e., N=*false*). By a positive operation we mean that information is either added to an EPD, $D*$, by an insert operation or positive information is to be retrieved from $D*$ if P=*true* (i.e., a user has requested to know whether some information is true in the EPD).

The '$-$' symbol in the triple of arguments $(N, -, P)$ in the *holds* predicate is used to denote that N is a negative operation which has the positive counterpart P. The positive counterpart P of N when N=*delete* is *insert* (i.e., P=*insert*); the positive counterpart P of N when N=*false* is *true* (i.e., P=*true*). By a negative operation we mean that information is either removed from $D*$ by a delete operation or negative information is to be retrieved if N=*false* (i.e., a user has requested to know whether some information is false in the EPD).

It follows from this discussion that the reading of the $holds(U, P, +, N, I)$ clause in an EPD that protects I in the case of TYPE 1 information is:

- $P = insert$ case: a user U inserts a ground instance of I into $D*$ if U is permitted to insert this instance of I into $D*$ to produce the updated state $\Delta D*$ of $D*$ and if U inserts (i.e., makes true) all Ai literals ($i \in \{1, .., m\}$) in U's authorized view of $D*$ and deletes (i.e., makes false) all Bj literals ($j \in \{1, .., n\}$) in U's authorized view of $D*$.
- $P = true$ case: a user U can know that a ground instance of I is true in $D*$ if U is permitted to know that this instance of I is true in $D*$ and U can know that each Ai literal ($i \in \{1, .., m\}$) is true in $D*$ and Ai is true in U's authorized view of $D*$, and U can know that each Bj literal ($j \in \{1, .., n\}$) is false in $D*$ and Bj is false in U's authorized view of $D*$.

4.2 Representing TYPE2 IDB Information in EPDs

To protect $I \leftarrow A1, \ldots, Am, not\ B1, \ldots, not\ Bn$ in the case of TYPE 2 information, I is expressed in an EPD thus:

$$
\begin{array}{l}
holds(U, N, -, P, I) \leftarrow permitted(U, N, I), holds(U, N, -, P, A1). \\
\ldots \\
holds(U, N, -, P, I) \leftarrow permitted(U, N, I), holds(U, N, -, P, Am). \\
holds(U, N, -, P, I) \leftarrow permitted(U, N, I), holds(U, P, +, N, B1). \\
\ldots \\
holds(U, N, -, P, I) \leftarrow permitted(U, N, I), holds(U, P, +, N, Bn).
\end{array}
$$

As with TYPE 1 information, the N variable refers to a "negative" operation which has the P operation as its positive counterpart. In the case of a user's request to delete a ground instance of I from an EPD $D*$, N=*delete* and P=*insert*. In the case when a user's request is to know whether I is false in $D*$, N=*false* and P=*true*.

The reading of $holds(U, N, -, P, I)$ clauses in the case of TYPE 2 information is:

- $N = delete$ case: a user U deletes a ground instance of I from $D*$ if U is permitted to delete I from $D*$ to produce the updated state $\Delta D*$ of $D*$ and if U deletes (i.e., makes false) some Ai literal ($i \in \{1, .., m\}$) in U's authorized view of $D*$ or U inserts (i.e., makes true) some Bj literal ($j \in \{1, .., n\}$) in U's authorized view of $D*$.
- $N = false$ case: a user U knows that a ground instance of I is false in $D*$ if U is permitted to know that this instance of I is false in $D*$, and U can know that some Ai literal ($i \in \{1, .., m\}$) is false in $D*$ and Ai is false in U's authorized view of $D*$ or U can know that some Bj literal ($j \in \{1, .., n\}$) is true in $D*$ and Bj is true in U's authorized view of $D*$.

4.3 Representing TYPE 1 EDB Information in EPDs

To express the protection of an arbitrary n-place EDB predicate, E, from unauthorized reads of positive information, a clause of the following form is used in

an EPD (where Xi $(i = (1..n))$ is a variable):

$$holds(U, true, +, false, E(X1, \ldots, Xn)) \leftarrow permitted(U, true, E(X1, \ldots, Xn)), \\ E(X1, \ldots, Xn).$$

That is, a ground instance of E is true in an EPD, $D*$, as far as a user U is concerned if U is authorized to know that this instance of E is true in $D*$, and the instance of E is true in $D*$.

For the EDB predicate E to be protected from an insert request on $D*$, a clause of the following form is included in $D*$:

$$holds(U, insert, +, delete, E(X1, .., Xn)) \leftarrow permitted(U, insert, E(X1, .., Xn)).$$

That is, a ground instance of E may be inserted into $D*$ in response to U's insert request if U is authorized to insert this instance into $D*$. Notice that $D*$ is not accessed when evaluating insert requests. As we will see, this ensures that all candidate changes to $D*$ that satisfy an insert request can be identified.

4.4 Representing TYPE 2 EDB Information in EPDs

The clause protecting E from unauthorized reads of negative information takes the form:

$$holds(U, false, -, true, E(X1, \ldots, Xn)) \leftarrow permitted(U, false, E(X1, \ldots, Xn)), \\ not\ E(X1, \ldots, Xn).$$

That is, a ground instance of E is false in an EPD, $D*$, as far as a user U is concerned if U is authorized to know that this instance of E is false in $D*$, and the instance of E is false in $D*$.

For the EDB predicate E to be protected from a delete request in $D*$, the required form of clause in $D*$ is:

$$holds(U, delete, -, insert, E(X1, \ldots, Xn)) \leftarrow permitted(U, delete, \\ E(X1, \ldots, Xn)).$$

That is, a ground instance of E may be deleted from an EPD, $D*$, by a user U if U is authorized to delete this instance of E from $D*$. As in the case of insertion, $D*$ is not accessed when delete requests are processed; this makes it possible to compute all candidate changes to $D*$ that will satisfy a delete request.

4.5 Representing EPDs

The ground assertions in the EDB of a deductive database D do not change when D is written in its EPD form, $D*$. Hence, $EDB(D*)$ is omitted in the

example EPD that follows. Note also that we use structured terms in our example EPD merely to simplify its expression. An equivalent function-free ("flattened") form of an EPD would be used in practice (to reduce term unification costs in computations).

Example 1 *Let* $D_1 = \{s(X) \leftarrow not\ q(X), r(X); r(b) \leftarrow\}$ *where* q *and* r *are* EDB *predicates, and ';' separates clauses. The EPD form of* D_1, D_1*, *is:*

$$holds(U, P, +, N, s(X)) \leftarrow permitted(U, P, s(X)), holds(U, N, -, P, q(X)), holds(U, P, +, N, r(X)).$$
$$holds(U, N, -, P, s(X)) \leftarrow permitted(U, N, s(X)), holds(U, P, +, N, q(X)).$$
$$holds(U, N, -, P, s(X)) \leftarrow permitted(U, N, s(X)), holds(U, N, -, P, r(X)).$$
$$holds(U, true, +, false, q(X)) \leftarrow permitted(U, true, q(X)), q(X).$$
$$holds(U, false, -, true, q(X)) \leftarrow permitted(U, false, q(X)), not\ q(X).$$
$$holds(U, true, +, false, r(X)) \leftarrow permitted(U, true, r(X)), r(X).$$
$$holds(U, false, -, true, r(X)) \leftarrow permitted(U, false, r(X)), not\ r(X).$$
$$holds(U, insert, +, delete, q(X)) \leftarrow permitted(U, insert, q(X)).$$
$$holds(U, delete, -, insert, q(X)) \leftarrow permitted(U, delete, q(X)).$$
$$holds(U, insert, +, delete, r(X)) \leftarrow permitted(U, insert, r(X)).$$
$$holds(U, delete, -, insert, r(X)) \leftarrow permitted(U, delete, r(X)).$$

In D_1*, the majority of clauses are required to represent the protection of $EDB(D_1*)$. However, the uniform nature of the EDB of a deductive database (i.e., the EDB is a set of ground atomic assertions) makes it possible to express the protection of EDB predicates in any EPD by using the following four clauses (in which O is a variable):

$$holds(U, insert, +, delete, O) \leftarrow permitted(U, insert, O).$$
$$holds(U, delete, -, insert, O) \leftarrow permitted(U, delete, O).$$
$$holds(U, false, -, true, O) \leftarrow permitted(U, false, O), not\ O.$$
$$holds(U, true, +, false, O) \leftarrow permitted(U, true, O), O.$$

Assuming that these four clauses are included in an EPD, it follows that D_1* may be (very simply) expressed thus:

$$holds(U, P, +, N, s(X)) \leftarrow permitted(U, P, s(X)), holds(U, N, -, P, q(X)), holds(U, P, +, N, r(X)).$$
$$holds(U, N, -, P, s(X)) \leftarrow permitted(U, N, s(X)), holds(U, P, +, N, q(X)).$$
$$holds(U, N, -, P, s(X)) \leftarrow permitted(U, N, s(X)), holds(U, N, -, P, r(X)).$$

It should also be noted that $D*$ is always stratified and that $IDB(D*)$ is always definite.

5 EPDs: Computational Issues

Since $D* \cup S*$, is expressed using a set of stratified clauses, it follows that a restricted form of SLG-resolution may be used to evaluate access requests (i.e.,

holds goal clauses) on $D* \cup S*$. As we will see, retrieval and update requests may be evaluated in a uniform way and, in the case of updates, a set of changes to $EDB(D*)$ is computed that will satisfy a user's authorized modification request.

5.1 Evaluating Retrieval Requests on EPDs

When a user u requests to retrieve positive instances of an n-place predicate p which are logical consequences of an EPD, $D*$, the goal clause that is evaluated, by using SLG-resolution on $D* \cup S*$, is of the following form:[2]

$$holds(u, true, +, false, p(t1, \ldots, tn)) \leftarrow holds(u, true, +, false, p(t1, \ldots, tn))$$

When the user u requests to know whether a ground instance of p is false in $D*$, the goal clause that is evaluated, by using SLG-resolution on $D* \cup S*$, is of the form:

$$holds(u, false, -, true, p(t1, \ldots, tn)) \leftarrow holds(u, false, -, true, p(t1, \ldots, tn))$$

A goal clause may be expressed directly on $D*$ by the user u or it may be set up by a simple routine that is used via a GUI. The identifier of the user posing the query on $D*$ will be extracted from the user login details (if a GUI is used), and will be checked against the login information if a user is permitted to pose a query directly on $D*$.

Example 2 *(Evaluating Retrieval Requests on EPDs) Consider the EPD, D_1*, from Example 1 and the following $RBAC_{H2A}$ program, S_1* on D_1*:*

$$S_1* = \{ura(bob, r1); pra(delete, q(X), r1); pra(insert, r(b), r1);$$
$$pra(A, s(X), r1); pra(A, r(a), r2); pra(insert, q(a), r2); ds(r1, r2)\}$$

Suppose that Bob wishes to know all instances of the 1-place predicate s that are true in D_1*. In this case, the goal clause that is evaluated by using SLG-resolution with respect to $D_1* \cup S_1*$ is:

$$holds(bob, true, +, false, s(X)) \leftarrow holds(bob, true, +, false, s(X))$$

Since Bob is assigned to the role $r1$ and $pra(A, s(X), r1)$ is in S_1* it follows that Bob has all of the authorizations that are necessary to know that any instance of s is true in D_1*. Since Bob is authorized to know that $s(b)$ is true in D_1* and $q(b)$ is false and $r(b)$ is true in Bob's authorized view of $D*$, it follows that $s(b)$ is true in D_1*. Hence, the query above will succeed by SLG-resolution and will generate the answer $X = b$.

[2] Here ti ($i \in \{1, .., n\}$) is a term and the goal clause is of the form $Q \leftarrow Q$.

5.2 Evaluating Update Requests on EPDs

The semantics of updates we implement for an EPD, $D*$, may be expressed thus (where $PM(\Delta D* \cup S*)$ denotes the perfect model of $\Delta D* \cup S*$):

Proposition 2 *(Insertion) A user u inserts object o into $D*$ iff $PM(\Delta D* \cup S*) \models holds(u, true, +, false, o)$.*

Proposition 3 *(Deletion) A user u deletes object o from $D*$ iff $PM(\Delta D* \cup S*) \models holds(u, false, -, true, o)$.*

That is, a request by a user u to insert o into $D*$ will be satisfied in the updated state $\Delta D*$ of $D*$ iff o is true in u's authorized view of $\Delta D* \cup S*$. Similarly, a request by u to delete o from $D*$ will be satisfied in $\Delta D*$ iff o is false in u's authorized view of $\Delta D* \cup S*$. For this semantics we require that update access implies read access. The following pair of stratified clauses may be included in $D*$ to specify that insert (delete) privilege on object O implies the privilege to know that O is true (false) in $\Delta D*$: $pra(read, O, R) \leftarrow pra(insert, O, R); pra(true, O, R) \leftarrow pra(false, O, R)$.

When a user u requests the insertion of an object o (a ground atomic formula) into an EPD, $D*$, protected by an $RBAC_{H2A}$ program, $S*$, the goal clause that is evaluated on $D* \cup S*$ by using SLG-resolution is:

$$holds(u, insert, +, delete, o) \leftarrow holds(u, insert, +, delete, o)$$

In the case of a request by u to delete o from $D*$, the goal clause that is evaluated with respect to $D* \cup S*$ is a ground instance of a clause of the form:

$$holds(u, delete, -, insert, o) \leftarrow holds(u, delete, -, insert, o)$$

As in the case of retrievals, an end user u may either specify a $holds(u, insert, +, delete, o)$ ($holds(u, delete, -, insert, o)$) goal clause directly on $D*$ or the goal clause may be set up via a GUI.

If E is an EDB predicate and a ground instance of $permitted(u, insert, E(x1, \ldots, xn))$ is an *SLG-resolvent* [8] in an SLG-derivation on $D* \cup S*$ that generates the answer clause $holds(u, insert, +, delete, o) \leftarrow$ or $holds(u, delete, -, insert, o) \leftarrow$ then inserting the instance of $E(x1, \ldots, xn)$ into $EDB(D*)$ is part of a candidate change transaction on $D*$ that may be performed to satisfy u's update request. Similarly, if E is an EDB predicate and $permitted(u, delete, E(x1, \ldots, xn))$ is a ground instance of an SLG-resolvent in an SLG-derivation on $D* \cup S*$ that generates the answer clause $holds(u, delete, -, insert, o) \leftarrow$ or $holds(u, insert, +, delete, o) \leftarrow$ then physically deleting the instance of $E(x1, \ldots, xn)$ from $EDB(D*)$ is part of a candidate change transaction on $D*$ that may be performed to satisfy u's update request.

Each SLG-derivation on $D* \cup S*$ that terminates with the answer clause $holds(u, insert, +, delete, o) \leftarrow$ (in the case of an insert request) or $holds(u, delete, -, insert, o) \leftarrow$ (in the case of a delete request) gives a candidate set of authorized insert and delete operations

on atoms in $EDB(D*)$ that will satisfy u's update request on $D*$. The changes required to $EDB(D*)$ are identified from the conjunction of $permitted(u, insert, P_i)$ and $permitted(u, delete, Q_j)$ subgoals involved in generating the answer clause. That is, if $permitted(u, insert, P_1) \wedge permitted(u, insert, P_2) \wedge \ldots \wedge permitted(u, insert, P_m) \wedge permitted(u, delete, Q_1) \wedge permitted(u, delete, Q_2) \wedge \ldots \wedge permitted(u, delete, Q_n)$ is the conjunction of *permitted* conditions that appear in an SLG-derivation on $D* \cup S*$ that produces an answer clause $holds(u, insert, +, delete, o) \leftarrow$ or $holds(u, delete, -, insert, o) \leftarrow$, and P_i $(i \in \{1, .., m\})$ and Q_j $(j \in \{1, .., n\})$ are EDB predicates then U's change request on $D*$ is authorized and the authorized modification that is required on $D*$ to satisfy U's change request is to insert P_1 and P_2 and ... and P_m into $EDB(D*)$ and to delete Q_1 and Q_2 and ... and Q_n from $EDB(D*)$.

It follows from the discussion above that there will be k sets of candidate change transactions iff there are k successful SLG-derivations for the evaluation of U's update request with respect to $D* \cup S*$. An update routine will be used to select a preferred change transaction, T, from the set of candidate change transactions on an EPD.[3] Moreover, the update routine will perform the inserts and deletes in T on $EDB(D*)$ to satisfy U's authorized update request.[4] Details of the implementation of the update routine can be found in [4].

If the SLG-derivation for a $holds(u, insert, +, delete, o)$ or $holds(u, delete, -, insert, o)$ goal clause on $D* \cup S*$ is failed then either the change request is satisfiable with respect to $EDB(D*)$ but U is not authorized to make it (i.e., $S*$ prevents U making the changes to $D*$ to produce $EDB(\Delta D*)$) or the update request cannot be satisfied by modifying $EDB(D*)$ alone.

Example 3 *(Satisfying Update Requests on EPDs) Consider again $D_1* \cup S_1*$ from Examples 1 and 2, and suppose that Bob wishes to delete $s(b)$ from $D_1* \cup S_1*$. The update request may be expressed thus:*

$$holds(bob, delete, -, insert, s(b)) \leftarrow holds(bob, delete, -, insert, s(b))$$

The goal clause above has two successful SLG-derivations. In one of the successful derivations, $permitted(bob, insert, q(b))$ is a subgoal; in the other successful derivation $permitted(bob, delete, r(b))$ is a subgoal. Hence, since p and q are EDB predicates, inserting $q(b)$ or deleting $r(b)$ are the candidate authorized changes to D_1* that may be performed to satisfy Bob's delete request.

The following general results apply to our update method (see [5] for proofs):

Theorem 2 *If $holds(u, insert, +, delete, o) \leftarrow$ is an answer clause by SLG-resolution on $D* \cup S*$ then $\Delta D* \cup S* \vdash_{SLG} holds(u, true, +, false, o)$.*

[3] The issues relating to the choice of the preferred update when a unique option does not exist is a *belief revision problem* [12] and is not addressed in this paper.

[4] Provided that these inserts/deletes satisfy the integrity constraints on $D*$.

Theorem 3 *If* $holds(u, delete, -, insert, O) \leftarrow$ *is an answer clause by SLG-resolution on* $D* \cup S*$ *then* $\Delta D* \cup S* \vdash_{SLG} holds(u, false, -, true, o)$.

It should also be noted that none of the clauses in D_1* contain a local variable. If clauses in $D*$ do contain local variables then *view update* [10] and belief revision problems may arise. However, these problems are not specific to EPDs, and their treatment is well beyond the scope of this paper.

6 EPDs: Performance Measures

We have conducted a number of performance tests on EPDs. The main bulk of these tests have involved evaluating a variety of recursive queries on EPDs, and performing queries which involve expensive join operations. The deductive databases that we have used in our testing have included between 25000 and 30000 EDB assertions that are relevant to the computations involved in our testing. We have also used some reasonably-sized $RBAC_{H2A}$ programs. These programs have included a 312 role $RBAC_{H2A}$ role hierarchy that has been represented using a set of ground assertions to represent all pairs of roles in the *senior_to* relation.

The EPDs that we have used for testing have been implemented using the *XSB* system (version 2.1) [19]. Our tests have been performed on a Sun Sparc machine running Solaris. To measure the performance of query evaluation, we have used the built-in *XSB* predicate *statistics*.

The results of our testing on EPDs have revealed an average 8-10 per cent increase in the time taken to process our test queries when a user has complete access to all predicates that are involved in the query evaluation. In contrast, when a user does not have full access to all predicates relevant to answering the query, the queries on $D* \cup S*$ can often be performed *more* efficiently than the same query run with respect to D.

The negative result that we have observed is perhaps not too surprising since in this case the $RBAC_{H2A}$ program does not constrain the search space of solutions; the $RBAC_{H2A}$ program simply adds to the processing overheads. The positive result is interesting since it suggests that, despite the increased size of program involved, the constraints on access defined in $S*$ may reduce the amount of computation that is involved.

In addition to helping to reduce the costs of query evaluation, access control information may also be used to reduce the number of candidate change transactions which are generated to satisfy authorized update requests.

7 Conclusions and Further Work

We have shown how deductive databases may be protected from unauthorized retrievals of positive and (implicit) negative information, and from the unauthorized insertion and deletion of atomic formulae by using logic programs. We have also demonstrated how retrievals of positive information and insertions

may be treated in a uniform way, and how retrievals of negative information and deletions can be similarly treated. Stratified logic programs are sufficient to implement EPDs and access control information may be exploited to help to optimize goal evaluation on EPDs. No special computational methods or language features are required for EPDs, and access control information can be seamlessly expressed on a deductive database by using a small number of clauses.

In future work, we intend to consider how our work on temporal authorization models [3] can be incorporated into our representation of EPDs, how extended forms of $RBAC_{H2A}$ program may be used for protecting information in EPDs, and how classical problems, such as query equivalence, change for EPDs.

References

1. Barker, S., Data Protection by Logic Programming, *1st International Conference on Computational Logic*, LNAI 1861, 1300-1313, Springer, 2000. 58
2. Barker, S., Secure Deductive Databases, *3rd Inat. Symp. on Practical Applications of Declarative Languages (PADL'01)*, LNCS 1990, Springer, 123-137, 2001. 55, 56
3. Barker, S., $TRBAC_N$: A Temporal Authorization Model, *Proc. MMMANCS International Workshop on Network Security*, LNCS 2052, 178-188, Springer, 2001. 68
4. Barker. S., and Douglas, P., Secure Web Access, *in preparation.* 66
5. Barker. S., Extended Protected Databases, *in preparation.* 66
6. Bonatti, P., Kraus, S., and Subrahmanian, V., Foundations of Secure Deductive Databases, *IEEE Trans. on Knowledge and Data Engineering*, 7, 3, 406-422, 1995. 55
7. Castano, S., Fugini, M., Martella, G., and Samarati, P., *Database Security*, Addison-Wesley, 1995. 55, 57
8. Chen, W., and Warren, D., Tabled Evaluation with Delaying for General Logic Programs, *J. ACM*, 43(1), 20-74, 1996. 57, 58, 65
9. Cuppens, F., and Demolombe, R., A Modal Logical Framework for Security Policies, *ISMIS'97*, 1997. 55
10. Date, C., *An Introduction to Database Systems*, Addison-Wesley, 2000. 67
11. Ferraiolo, D., Gilbert, D., and Lynch, N., An Examination of Federal and Commercial Access Control Policy Needs, *Proc. 16th NIST-NSA National Computer Security Conference*, 107-116, 1993. 54
12. Gardenfors, P., *Knowledge in Flux: Modeling the Dynamics of Epistemic States*, MIT Press, 1988. 66
13. Hogger, C., *Foundations of Logic Programming*, Oxford, 1990. 55
14. Jajodia, S., Samarati, P., and Subrahmanian, V., A Logical Language for Expressing Authorizations, *Proc. IEEE Symp. on Security and Privacy*, 94-107, 1997. 55
15. Jamil, H., Belief Reasoning in MLS Deductive Databases, *ACM SIGMOD'99*, 109-120, 1999. 55
16. LLoyd, J., *Foundations of Logic Programming*, Springer, 1987. 59
17. Minker, J., Logic and Databases: A 20 Year Retrospective, *1st International Workshop on Logic in Databases*, LNCS 1154, 3-57, Springer, 1996. 54
18. Przymusinski, T., Perfect Model Semantics, *Proc. 5th ICLP*, MIT Press, 1081-1096, 1988. 58
19. Sagonas, K., Swift, T., Warren, D., Freire, J., Rao, P., The XSB System, Version 2.0, Programmer's Manual, 1999. 67

20. Sandhu, R., Ferraiolo, D., and Kuhn, R., The NIST Model for Role-Based Access Control: Towards a Unified Standard, *Proc. 4th ACM Workshop on Role-Based Access Control*, 47-61, 2000. 54, 55, 57
21. Ullman, J., *Principles of Database and Knowledge-Base Systems: Volume 1*, Computer Science Press, 1990. 58
22. Van Gelder, A., Ross, K., and Schlipf, J., The Well-Founded Semantics for General Logic Programs, *J. ACM*, 38(3), 620-650, 1991. 57

Reasoning about Actions with CHRs and Finite Domain Constraints

Michael Thielscher*

Dresden University of Technology
mit@inf.tu-dresden.de

Abstract. We present a CLP-based approach to reasoning about actions in the presence of incomplete states. Constraints expressing negative and disjunctive state knowledge are processed by a set of special Constraint Handling Rules. In turn, these rules reduce to standard finite domain constraints when handling variable arguments of single state components. Correctness of the approach is proved against the general action theory of the Fluent Calculus. The constraint solver is used as the kernel of a high-level programming language for agents that reason and plan. Experiments have shown that the constraint solver exhibits excellent computational behavior and scales up well.

1 Introduction

One of the most challenging and promising goals of Artificial Intelligence research is the design of autonomous agents, including robots, that explore partially known environments and that are able to act sensibly under incomplete information. To attain this goal, the paradigm of Cognitive Robotics [5] is to endow agents with the high-level cognitive capabilities of reasoning and planning: Exploring their environment, agents need to reason when they interpret sensor information, memorize it, and draw inferences from combined sensor data. Acting under incomplete information, agents employ their reasoning facilities to ensure that they are acting cautiously, and they plan ahead some of their actions with a specific goal in mind. To this end, intelligent agents form a mental model of their environment, which they constantly update to reflect the changes they have effected and the sensor information they have acquired.

Having agents maintain an internal world model is necessary if we want them to choose their actions not only on the basis of the current status of their sensors but also by taking into account what they have previously observed or done. Moreover, the ability to reason about sensor information is necessary if properties of the environment can only indirectly be observed and require the agent to combine observations made at different stages. The ability to plan allows an agent to first calculate the effect of different action sequences in order to help it choosing one that is appropriate under the current circumstances.

* Parts of the work reported in this paper have been carried out while the author was a visiting researcher at the University of New South Wales in Sydney, Australia.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 70–84, 2002.
© Springer-Verlag Berlin Heidelberg 2002

While standard programming languages such as Java do not provide general reasoning facilities for agents, logic programming constitutes the ideal paradigm for designing agents that are capable of reasoning about their actions [9]. Examples of existing LP-systems deriving from general action theories are GOLOG [6,8], based on the Situation Calculus [7], or the robot control language developed in [10], based on the Event Calculus [4]. However, a disadvantage of both these systems is that knowledge of the current state is represented indirectly via the initial conditions and the actions which the agent has performed up to a point. As a consequence, each time a condition is evaluated in an agent program, the entire history of actions is involved in the computation. This requires ever increasing computational effort as the agent proceeds, so that this concept does not scale up well to long-term agent control [13].

An explicit state representation being a fundamental concept in the Fluent Calculus [11], this representation formalism offers an alternative theory as the formal underpinnings for a high-level agent programming method. In this paper, we present a CLP approach to reasoning about actions which implements the Fluent Calculus. Incomplete states are represented as *open* lists of state properties, that is, lists with a variable tail. Negative and disjunctive state knowledge is encoded by *constraints*. We present a set of so-called Constraint Handling Rules (CHRs) [2] for combining and simplifying these constraints. In turn, these rules reduce to standard finite domain constraints when handling variable arguments of single state components. Based on their declarative interpretation, our CHRs are verified against the foundational axioms of the Fluent Calculus. The constraint solver is used as the kernel of the high-level programming language FLUX (for: *Fluent Executor*) which allows the design of intelligent agents that reason and plan on the basis of the Fluent Calculus [12]. Studies have shown that FLUX and in particular the constraint solver scale up well [13].

The paper is organized as follows: In Section 2, we recapitulate the basic notions and notations of the Fluent Calculus as the underlying theory for our CLP-based approach to reasoning about actions. In Section 3, we present a set of CHRs for constraints expressing negative and disjunctive state knowledge, and we prove their correctness wrt. the foundational axioms of the Fluent Calculus. In Section 4, we embed the constraint solver into a logic program for reasoning about actions, which, too, is verified against the underlying semantics of the Fluent Calculus. In Section 5, we give a summary of studies showing the computational merits of our approach. We conclude in Section 6.

The constraint solver, the general FLUX system, the example agent program, and the accompanying papers all are available for download at our web site `http://fluxagent.org`.

2 Reasoning about States with the Fluent Calculus

Throughout the paper, we will use the following example of an agent in a dynamic environment: Consider a cleaning robot which, in the evening, has to empty the waste bins in the alley and rooms of the floor of an office building. The robot

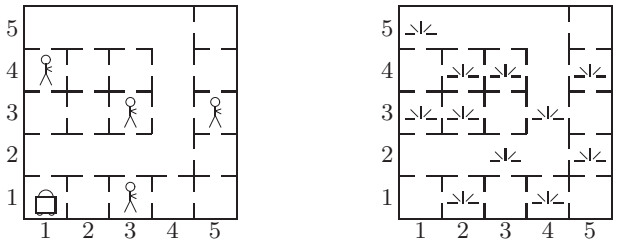


Fig. 1. Layout of a sample office floor and a scenario in which four offices are occupied. In the right hand side the locations are depicted in which the robot senses light

shall not, however, disturb anyone working in late. The robot is equipped with a light sensor which is activated whenever the robot is adjacent to a room that is occupied, without being able to tell which direction the light comes from. An instance of this problem is depicted in Fig. 1. The task is to program the cleaning robot so as to empty as many bins as possible without risking to burst into an occupied office. This problem illustrates two challenges raised by incomplete state knowledge: Agents have to act cautiously, and they need to interpret and logically combine sensor information acquired over time.

The Fluent Calculus is an axiomatic theory of actions that provides the formal underpinnings for agents to reason about their actions [11]. Formally, it is a many-sorted predicate logic language with four standard sorts for actions and situations (as in the Situation Calculus) and for fluents (i.e., atomic state properties) and states. For the cleaning robot domain, for example, we will use these four fluents (i.e., mappings into the sort FLUENT): $At(x, y)$, representing that the robot is at (x, y); $Facing(d)$, representing that the robot faces direction $d \in \{1, \ldots, 4\}$ (denoting, resp., north, east, south, and west); $Cleaned(x, y)$, representing that the waste bin at (x, y) has been emptied; and $Occupied(x, y)$, representing that (x, y) is occupied. We make the standard assumption of uniqueness-of-names, $UNA[At, Faces, Cleaned, Occupied]$.[1]

States are built up from fluents (as atomic states) and their conjunction, using the function $\circ$: STATE $\times$ STATE $\mapsto$ STATE along with the constant $\emptyset$: STATE denoting the empty state. For example, the term $At(1, 1) \circ (Facing(1) \circ z)$ represents a state in which the robot is in square $(1, 1)$ facing north while other fluents may hold, too, summarized in the variable sub-state z.

A fundamental notion is that of a fluent to *hold* in a state. Fluent f is said to hold in state z just in case z can be decomposed into two states one of which is the singleton f. For notational convenience, we introduce the macro $Holds(f, z)$ as an abbreviation for the corresponding equational formula:

$$Holds(f, z) \stackrel{\text{def}}{=} (\exists z')\, z = f \circ z' \tag{1}$$

[1] $UNA[h_1, \ldots, h_n] \stackrel{\text{def}}{=} \bigwedge_{i \neq j} h_i(\boldsymbol{x}) \neq h_j(\boldsymbol{y}) \wedge \bigwedge_i [h_i(\boldsymbol{x}) = h_i(\boldsymbol{y}) \supset \boldsymbol{x} = \boldsymbol{y}]$.

This definition is accompanied by the following foundational axioms of the Fluent Calculus, which constitute a special theory of equality of state terms.

Definition 1. *Assume a signature which includes the sorts* FLUENT *and* STATE *such that* FLUENT *is a sub-sort of* STATE*, along with the functions* $\circ, \emptyset$ *of sorts as above. The* foundational axioms Σ_{state} of the Fluent Calculus *are:*[2]

1. *Associativity, commutativity, idempotence, and unit element,*

$$\begin{array}{ll} (z_1 \circ z_2) \circ z_3 = z_1 \circ (z_2 \circ z_3) & z \circ z = z \\ z_1 \circ z_2 = z_2 \circ z_1 & z \circ \emptyset = z \end{array} \tag{2}$$

2. *Empty state axiom,*

$$\neg Holds(f, \emptyset) \tag{3}$$

3. *Irreducibility and decomposition,*

$$Holds(f_1, f) \supset f_1 = f \tag{4}$$

$$Holds(f, z_1 \circ z_2) \supset Holds(f, z_1) \vee Holds(f, z_2) \tag{5}$$

4. *State equivalence and existence of states,*

$$(\forall f)\,(Holds(f, z_1) \equiv Holds(f, z_2)) \supset z_1 = z_2 \tag{6}$$

$$(\forall P)(\exists z)(\forall f)\,(Holds(f, z) \equiv P(f)) \tag{7}$$

where P *is a second-order predicate variable of sort* FLUENT*.*

Axioms (2) essentially characterize "$\circ$" as the union operation with $\emptyset$ as the empty set of fluents. Associativity allows us to omit parentheses in nested applications of "$\circ$". Axiom (6) says that two states are equal if they contain the same fluents, and second-order axiom (7) guarantees the existence of a state for any combination of fluents.

The foundational axioms can be used to reason about incomplete state specifications and acquired sensor information. Consider, e.g., the definition of what it means for our cleaning robot to sense light in a square (x, y) in some state z:

$$\begin{array}{l} LightPerception(x, y, z) \equiv \\ \quad Holds(Occupied(x+1, y), z) \vee Holds(Occupied(x, y+1), z) \\ \quad \vee\ Holds(Occupied(x-1, y), z) \vee Holds(Occupied(x, y-1), z) \end{array} \tag{8}$$

Suppose that at the beginning the robot only knows that the following locations are not occupied: its home $(1, 1)$ (axiom (10) below), the squares in the alley (axiom (11) below), and any location outside the boundaries of the office floor (axioms (12),(13) below). Suppose further that the robot already went to

[2] Throughout the paper, free variables in formulas are assumed universally quantified. Variables of sorts FLUENT, STATE, ACTION, and SIT shall be denoted by the letters f, z, a, and s, resp. The function "$\circ$" is written in infix notation.

clean $(1,1)$, $(1,2)$, and $(1,3)$, sensing light in the last square only (c.f. Fig. 1). Thus the current state, ζ, is known to be

$$\zeta = At(1,3) \circ Facing(1) \circ Cleaned(1,1) \circ Cleaned(1,2) \circ Cleaned(1,3) \circ z \quad (9)$$

for some z, along with the following axioms:

$$\neg Holds(Occupied(1,1), z) \quad (10)$$
$$\neg Holds(Occupied(1,5), z) \wedge \ldots \wedge \neg Holds(Occupied(1,2), z) \quad (11)$$
$$(\forall x)\,(\neg Holds(Occupied(x,0), z) \wedge \neg Holds(Occupied(x,6), z)) \quad (12)$$
$$(\forall y)\,(\neg Holds(Occupied(0,y), z) \wedge \neg Holds(Occupied(6,y), z)) \quad (13)$$
$$\neg LightPerception(1,1,\zeta) \wedge \neg LightPerception(1,2,\zeta) \quad (14)$$
$$LightPerception(1,3,\zeta) \quad (15)$$

From (14) and (8), $\neg Holds(Occupied(2,1),\zeta) \wedge \neg Holds(Occupied(2,2),\zeta)$. With regard to (9), the foundational axioms of decomposition, (5), and irreducibility, (4), along with uniqueness-of-names imply

$$\neg Holds(Occupied(2,1), z) \wedge \neg Holds(Occupied(2,2), z) \quad (16)$$

On the other hand, (15) and (8) imply

$$Holds(Occupied(2,3),\zeta) \vee Holds(Occupied(1,4),\zeta)$$
$$\vee Holds(Occupied(0,3),\zeta) \vee Holds(Occupied(1,2),\zeta)$$

As above, with regard to (9), the foundational axioms of decomposition and irreducibility along with uniqueness-of-names imply

$$Holds(Occupied(2,3), z) \vee Holds(Occupied(1,4), z)$$
$$\vee Holds(Occupied(0,3), z) \vee Holds(Occupied(1,2), z)$$

From (13) and (11) it follows that

$$Holds(Occupied(2,3), z) \vee Holds(Occupied(1,4), z) \quad (17)$$

This disjunction cannot be reduced further, that is, at this stage the robot does not know whether the light in $(1,3)$ comes from office $(2,3)$ or $(1,4)$ (or both, for that matter). Suppose, therefore, the cautious robot goes back, turns east, and continues with cleaning $(2,2)$, which it knows to be unoccupied according to (16). Sensing no light there (c.f. Fig. 1), the new state ζ' is known to be $At(2,2) \circ Facing(2) \circ Cleaned(1,1) \circ Cleaned(1,2) \circ Cleaned(1,3) \circ Cleaned(2,2) \circ z$ for some z such that (10)–(13),(16),(17) along with $\neg LightPerception(2,2,\zeta')$. From (8), $\neg Holds(Occupied(2,3),\zeta')$; hence, decomposition and irreducibility along with uniqueness-of-names imply $\neg Holds(Occupied(2,3), z)$; hence by (17), $Holds(Occupied(1,4), z)$, that is, now the robot knows that $(1,4)$ is occupied.

3 Solving State Constraints

Based on the axiomatic foundation of the Fluent Calculus, in the following we develop a provably correct CLP-approach for reasoning about incomplete state specifications. To begin with, incomplete states are encoded by *open* lists of fluents (possibly containing variables):

```
Z = [F1,...,Fk | _]
```

It is assumed that the arguments of fluents are encoded by natural or rational numbers, which enables the use of a standard arithmetic solver for constraints on partially known arguments. Negative and disjunctive state knowledge is expressed by the following *state constraints*:

constraint	semantics
`not_holds(F,Z)`	$\neg Holds(f, z)$
`not_holds_all(F,Z)`	$(\forall \boldsymbol{x})\, \neg Holds(f, z)$, where $\boldsymbol{x}$ variables in f
`or([F1,...,Fn],Z)`	$\bigvee_{i=1}^{n} Holds(f_i, z)$

The state constraints have been carefully designed so as to be sufficiently expressive while allowing for efficient constraint solving. An auxiliary constraint `duplicate_free(Z)` is used to stipulate that a list of fluents contains no multiple occurrences, thus reflecting the foundational axiom of idempotence of "$\circ$" in the Fluent Calculus. As an example, the following clause encodes the specification of state ζ of Section 2 (c.f. (9) and (8), resp.):

```
zeta(Zeta) :-
  Zeta = [at(1,3),facing(1),cleaned(1,1),cleaned(1,2),cleaned(1,3) | Z],
  not_holds(occupied(1,1),Z),
  not_holds(occupied(1,5),Z), ..., not_holds(occupied(1,2),Z),
  not_holds_all(occupied(_,0),Z), not_holds_all(occupied(_,6),Z),
  not_holds_all(occupied(0,_),Z), not_holds_all(occupied(6,_),Z),
  light_perception(1,1,false,Zeta), light_perception(1,2,false,Zeta),
  light_perception(1,3,true,Zeta),
  duplicate_free(Zeta).

light_perception(X,Y,Percept,Z) :-
  XE#=X+1, XW#=X-1, YN#=Y+1, YS#=Y-1,
  ( Percept=false,
      not_holds(occupied(XE,Y),Z), not_holds(occupied(X,YN),Z),
      not_holds(occupied(XW,Y),Z), not_holds(occupied(X,YS),Z)
    ; Percept=true, or([occupied(XE,Y),occupied(X,YN),
                        occupied(XW,Y),occupied(X,YS)],Z) ).
```

Here and in the following we employ a standard constraint domain, namely, that of finite domains, which includes arithmetic constraints over rational numbers using the equality, inequality, and ordering predicates `#=`,`#<`,`#>` along with the standard functions `+`,`-`,`*`; range constraints (written `X::[`a`..`b`]`); and logical combinations using `#/\` and `#\/` for conjunction and disjunction, resp.

Our approach is based on so-called Constraint Handling Rules, which support the declarative programming of constraint solvers [2]. CHRs are of the form

```
H1,...,Hm <=> G1,...,Gk | B1,...,Bn.
```

where the *head* $H_1, \ldots, H_m$ are constraints ($m \geq 1$); the *guard* $G_1, \ldots, G_k$ are Prolog literals ($k \geq 0$); and the *body* $B_1, \ldots, B_n$ are constraints ($n \geq 0$). An empty guard is omitted; the empty body is denoted by *True*. The declarative interpretation of a CHR is given by the formula

$$(\forall \boldsymbol{x})\,(G_1 \wedge \ldots \wedge G_k \supset [H_1 \wedge \ldots \wedge H_m \equiv (\exists \boldsymbol{y})\,(B_1 \wedge \ldots \wedge B_n)])$$

where $\boldsymbol{x}$ are the variables in both guard and head and $\boldsymbol{y}$ are the variables which additionally occur in the body. The procedural interpretation of a CHR is given by a transition in a constraint store: If the head can be matched against elements of the constraint store and the guard can be derived, then the constraints of the head are replaced by the constraints of the body.

3.1 Handling Negation

Fig. 2 depicts the first part of the constraint solver, which contains the CHRs and auxiliary clauses for the two negation constraints and the auxiliary constraint on multiple occurrences. In the following, these rules are proved correct wrt. the foundational axioms of the Fluent Calculus.

To begin with, consider the auxiliary clauses, which define a finite domain constraint that expresses the inequality of two fluent terms. By `or_neq` inequality of two fluents with arguments $\texttt{ArgX} = [\texttt{X1}, \ldots, \texttt{Xn}]$ and $\texttt{ArgY} = [\texttt{Y1}, \ldots, \texttt{Yn}]$ is decomposed into the arithmetic constraint $\texttt{X1} \not= \texttt{Y1} \vee \ldots \vee \texttt{Xn} \not= \texttt{Yn}$. Two cases are distinguished depending on whether the variables in the first term are existentially or universally quantified. In the latter case, a simplified disjunction is generated, where the variables of the first fluent are discarded while possibly giving rise to dependencies among the arguments of the second fluent. E.g., $\texttt{neq_all}(\texttt{f}(_, \texttt{a}, _), \texttt{f}(\texttt{U}, \texttt{V}, \texttt{W}))$ reduces to $\texttt{a} \not= \texttt{W}$ and $\texttt{neq_all}(\texttt{f}(\texttt{X}, \texttt{X}, \texttt{X}), \texttt{f}(\texttt{U}, \texttt{V}, \texttt{W}))$ reduces to $\texttt{U} \not= \texttt{W} \vee \texttt{V} \not= \texttt{W}$. To formally capture the universal quantification, we define the notion of a *schematic* fluent $f = h(\boldsymbol{x}, \boldsymbol{r})$ where $\boldsymbol{x}$ denotes the variable arguments in f. The following observation implies the correctness of the constraints generated by the auxiliary clauses.

Observation 1 *Consider a set $\mathcal{F}$ of functions into sort* FLUENT. *Consider a fluent $f_1 = g(r_1, \ldots, r_m)$, a schematic fluent $f_2 = g(x_1, \ldots, x_k, r_{k+1}, \ldots, r_m)$, and a fluent $f = h(t_1, \ldots, t_n)$. Then*

1. *if $g \not= h$, then $UNA[\mathcal{F}] \models f_1 \not= f$ and $UNA[\mathcal{F}] \models (\forall \boldsymbol{x})\, f_2 \not= f$;*
2. *if $g = h$, then $m = n$ and $UNA[\mathcal{F}] \models f_1 \not= f \equiv r_1 \not= t_1 \vee \ldots \vee r_m \not= t_n$ and $UNA[\mathcal{F}] \models (\forall x)\,(f_2 \not= f \equiv r_{k+1} \not= t_{k+1} \vee \ldots \vee r_m \not= t_n \vee \bigvee_{\substack{i \neq j \\ x_i = x_j}} t_i \not= t_j)$.*

CHRs 1–4 for negation constraints can then be justified by the foundational axioms of the Fluent Calculus, as the following proposition shows.

```
not_holds(_,[])          <=> true.                                    %1
not_holds(F,[F1|Z])      <=> neq(F,F1), not_holds(F,Z).               %2
not_holds_all(_,[])      <=> true.                                    %3
not_holds_all(F,[F1|Z]) <=> neq_all(F,F1), not_holds_all(F,Z).       %4

not_holds_all(F,Z)~not_holds(G,Z)     <=> instance(G,F) | true. %5
not_holds_all(F,Z)~not_holds_all(G,Z) <=> instance(G,F) | true. %6

duplicate_free([])    <=> true.                                      %7
duplicate_free([F|Z]) <=> not_holds(F,Z), duplicate_free(Z).         %8

neq(F,F1)     :- or_neq(exists,F,F1).
neq_all(F,F1) :- or_neq(forall,F,F1).

or_neq(Q,Fx,Fy) :- Fx =.. [F|ArgX], Fy =.. [G|ArgY],
                   ( F=G -> or_neq(Q,ArgX,ArgY,D), call(D) ; true ).

or_neq(_,[],[],(0#\=0)).
or_neq(Q,[X|X1],[Y|Y1],D) :-
  or_neq(Q,X1,Y1,D1),
  ( Q=forall, var(X) ->
       ( binding(X,X1,Y1,YE) -> D=((Y#\=YE)#\/D1) ; D=D1 ) ;
    D=((X#\=Y)#\/D1) ).

binding(X,[X1|ArgX],[Y1|ArgY],Y) :- X==X1 -> Y=Y1
                                          ; binding(X,ArgX,ArgY,Y).
```

Fig. 2. CHRs for negation and multiple occurrences. The notation `H1 \ H2 <=> G | B` is an abbreviation for `H1,H2 <=> G | H1,B`

Proposition 1. *Σ_{state} entails,*

1. $\neg Holds(f, \emptyset)$*; and*
2. $\neg Holds(f, f_1 \circ z) \equiv f \not= f_1 \wedge \neg Holds(f, z)$.

Likewise, if $f = g(\boldsymbol{x}, \boldsymbol{r})$ is a schematic fluent, then Σ_{state} entails,

3. $(\forall \boldsymbol{x}) \neg Holds(f, \emptyset)$*; and*
4. $(\forall \boldsymbol{x}) \neg Holds(f, f_1 \circ z) \equiv (\forall \boldsymbol{x}) f \not= f_1 \wedge (\forall \boldsymbol{x}) \neg Holds(f, z)$.

Proof. Claim 1 follows by the empty state axiom. Regarding claim 2 we prove that $Holds(f, f_1 \circ z) \equiv f = f_1 \vee Holds(f, z)$. The "$\supset$" direction follows by foundational axioms (5) and (4). For the "$\subset$" direction, if $f = f_1$, then $f_1 \circ z = f \circ z$, hence $Holds(f, f_1 \circ z)$. Likewise, if $Holds(f, z)$, then $z = f \circ z'$ for some z', hence $f_1 \circ z = f_1 \circ f \circ z'$, hence $Holds(f, f_1 \circ z)$. The proof of 3 and 4 is similar.

Correctness of CHRs 5 and 6, which remove subsumed negative constraints, is obvious as $(\forall \boldsymbol{x}) \neg Holds(f_1, z)$ implies $\neg Holds(f_2, z)$ and $(\forall \boldsymbol{y}) \neg Holds(f_2, z)$,

```
or([F],Z) <=> F\=eq(_,_) | holds(F,Z).                                  %9
or(V,Z)   <=> \+ ( member(F,V),F\=eq(_,_) )                             %10
              | or_and_eq(V,D), call(D).
or(V,[])  <=> member(F,V,W), F\=eq(_,_) | or(W,[]).                     %11

or(V,Z) <=> member(eq(X,Y),V),                                          %12
            or_neq(exists,X,Y,D), \+ call(D) | true.
or(V,Z) <=> member(eq(X,Y),V,W),                                        %13
            \+ (and_eq(X,Y,D), call(D)) | or(W,Z).

not_holds(F,Z)~or(V,Z)      <=> member(G,V,W), F==G | or(W,Z).    %14
not_holds_all(F,Z)~or(V,Z) <=> member(G,V,W),                       %15
                                    instance(G,F) | or(W,Z).

or(V,[F|Z])   <=> or(V,[],[F|Z]).                                       %16
or(V,W,[F|Z]) <=> member(F1,V,V1), \+ F\=F1                             %17
                  | ( F1==F -> true ; F1=..[_|ArgX], F=..[_|ArgY],
                    or(V1, [eq(ArgX,ArgY),F1|W], [F|Z]) ).
or(V,W,[_|Z]) <=> append(V,W,V1), or(V1,Z).                             %18

and_eq([],[],(0#=0)).
and_eq([X|X1],[Y|Y1],D) :- and_eq(X1,Y1,D1), D=((X#=Y)#/\D1).

or_and_eq([],(0#\=0)).
or_and_eq([eq(X,Y)|Eq],(D1#\/D2)) :- or_and_eq(Eq,D1), and_eq(X,Y,D2).

member(X,[X|T],T).
member(X,[H|T],[H|T1]) :- member(X,T,T1).
```

Fig. 3. CHRs for the disjunctive constraint

resp., for a schematic fluent f_1 and a fluent f_2 such that $f_1\theta = f_2$ for some θ. Finally, CHRs 7 and 8 for the auxiliary constraint on multiple occurrences are correct since the empty list contains no duplicate elements and a non-empty list contains no duplicates iff the head does not occur in the tail and the tail itself is free of duplicates.

3.2 Handling Disjunction

Fig. 3 depicts the second part of the constraint solver, which contains the CHRs and auxiliary clauses for the disjunctive constraint. Internally, a disjunctive constraint may contain, besides fluents, atoms of the form $\mathtt{eq}(\boldsymbol{r}, \boldsymbol{t})$ where $\boldsymbol{r}$ and $\boldsymbol{t}$ are lists of equal length. Such a general disjunction $\mathtt{or}([\delta_1, \ldots, \delta_k], \mathtt{Z})$ means

$$\bigvee_{i=1}^{k} \begin{cases} \mathit{Holds}(\mathtt{F}, \mathtt{Z}) & \text{if } \delta_i \text{ is fluent } \mathtt{F} \\ \boldsymbol{x} = \boldsymbol{y} & \text{if } \delta_i \text{ is } \mathtt{eq}(\boldsymbol{x}, \boldsymbol{y}) \end{cases} \tag{18}$$

CHR 9 in Fig. 3 simplifies singleton disjunctions according to (18). CHR 10 reduces a pure equational disjunction to a finite domain constraint. Its correctness follows directly from (18), too. CHR 11 simplifies a disjunction applied to the empty state. It is justified by the empty state axiom, (3), which entails

$$[Holds(f, \emptyset) \vee \Psi] \equiv \Psi$$

for any formula Ψ. CHRs 12 and 13 apply to disjunctions which include a decided equality. If the equality is true, then the entire disjunction is true, else if the equality is false, then the disjunction gets simplified. Correctness follows from

$$\boldsymbol{x} = \boldsymbol{y} \supset [\boldsymbol{x} = \boldsymbol{y} \vee \Psi \equiv True] \quad \text{and} \quad \boldsymbol{x} \neq \boldsymbol{y} \supset [\boldsymbol{x} = \boldsymbol{y} \vee \Psi \equiv \Psi]$$

The next two CHRs, 14 and 15, constitute unit resolution steps. They are justified by

$$\neg Holds(f, z) \supset [Holds(f, z) \vee \Psi \equiv \Psi]$$
$$(\forall \boldsymbol{x})\, \neg Holds(f_1, z) \supset [Holds(f_2, z) \vee \Psi \equiv \Psi]$$

given that $f_1\theta = f_2$ for some θ,

Finally, CHRs 16-18 in Fig. 3 are used to propagate a disjunction through a non-variable state. Given the constraint $\texttt{or}(\delta, [\texttt{F} \mid \texttt{Z}])$, the basic idea is to infer all possible bindings of `F` with fluents in δ. The rules use the auxiliary constraint $\texttt{or}(\delta, \gamma, [\texttt{F} \mid \texttt{Z}])$ with the intended semantics $\texttt{or}(\delta, [\texttt{F} \mid \texttt{Z}]) \vee \texttt{or}(\gamma, \texttt{Z})$, that is, δ contains the fluents that have not yet been evaluated against the head `F` of the state list, while γ contains those fluents that have been evaluated. As an example, consider the constraint `or([f(a,V),f(W,b)],[f(X,Y)|Z])` which, upon being processed, yields

```
or([f(a,V),f(W,b),eq([a,V],[X,Y]),eq([W,b],[X,Y])], Z)
```

The rules are justified by the following proposition.

Proposition 2. *Consider a Fluent Calculus signature with a set $\mathcal{F}$ of functions into sort* FLUENT. *Foundational axioms Σ_{state} and uniqueness-of-names $UNA[\mathcal{F}]$ entail each of the following:*

1. $\Psi \equiv [\Psi \vee \bigvee_{i=1}^{0} \Psi_i]$;
2. $[Holds(f(\boldsymbol{x}), f(\boldsymbol{y}) \circ z) \vee \Psi_1] \vee \Psi_2 \equiv \Psi_1 \vee [\boldsymbol{x} = \boldsymbol{y} \vee Holds(f(\boldsymbol{x}), z) \vee \Psi_2]$;
3. *if* $\bigwedge_{i=1}^{n} f_i \neq f$, *then* $[\bigvee_{i=1}^{n} Holds(f_i, f \circ z) \vee \Psi] \equiv [\bigvee_{i=1}^{n} Holds(f_i, z) \vee \Psi]$.

Proof. Claim 1 is obvious. Claims 2 and 3 follow from the foundational axioms of decomposition and irreducibility.

This completes the constraint solver. As an example, running the specification from the beginning of this section results in

```
?- zeta(Zeta).

Zeta=[at(1,3),facing(1),cleaned(1,1),cleaned(1,2),cleaned(1,3) | Z]
Constraints:
or([occupied(1,4),occupied(2,3)], Z)
...
```

Adding the information that there is no light in $(2,2)$, the system is able to infer that $(4,1)$ must be occupied:

```
?- zeta(Zeta), light_perception(2,2,Zeta,false).

Zeta=[at(1,3),facing(1),cleaned(1,1),cleaned(1,2),cleaned(1,3),
      occupied(4,1) | Z]
Constraints:
not_holds(occupied(2,3), Z)
...
```

While the FLUX constraint system is sound, it may not enable agents to draw all conclusions that follow logically from a state specification because the underlying standard arithmetic solver trades completeness for efficiency. This is due to the fact that a conjunction or a disjunction is evaluated only if one of its atoms has been decided. The advantage of so doing is that the computational effort of evaluating a new constraint is *linear* in the size of the constraint store while a complete solver would require exponential time for this task.

4 Reasoning about Actions

In this section, we embed our constraint solver into a logic program for reasoning about the effects of actions based on the Fluent Calculus. Generalizing previous approaches [3,1], the Fluent Calculus provides a solution to the fundamental frame problem in the presence of incomplete states [11]. The solution is based on a rigorously axiomatic characterizations of addition and removal of (finitely many) fluents from incompletely specified states. The following definition introduces the macro equation $z_1 - \tau = z_2$ with the intended meaning that state z_2 is state z_1 minus the fluents in the finite state τ. The compound macro $z_2 = (z_1 - \vartheta^-) + \vartheta^+$ means that state z_2 is state z_1 minus the fluents in ϑ^- plus the fluents in ϑ^+:

$$
\begin{aligned}
&z_1 - \emptyset = z_2 \stackrel{\text{def}}{=} z_2 = z_1\\
&z_1 - f = z_2 \stackrel{\text{def}}{=} (z_2 = z_1 \vee z_2 \circ f = z_1) \wedge \neg \mathit{Holds}(f, z_2)\\
&z_1 - (f_1 \circ f_2 \circ \ldots \circ f_n) = z_2 \stackrel{\text{def}}{=} (\exists z)\,(z = z_1 - f \wedge z_2 = z - (f_2 \circ \ldots \circ f_n))\\
&(z_1 - \vartheta^-) + \vartheta^+ = z_2 \stackrel{\text{def}}{=} (\exists z)\,(z = z_1 - \vartheta^- \wedge z_2 = z \circ \vartheta^+)
\end{aligned}
$$

where both ϑ^+, ϑ^- are finitely many FLUENT terms connected by "$\circ$". The crucial item is the second one, which defines removal of a single fluent f using a case distinction: Either $z_1 - f$ equals z_1 (which applies in case $\neg \mathit{Holds}(f, z_1)$), or $z_1 - f$ plus f equals z_1 (which applies in case $\mathit{Holds}(f, z_1)$).

Fig. 4 depicts a set of clauses which encode the solution to the frame problem on the basis of the constraint solver for the Fluent Calculus. The program culminates in the predicate $\mathit{Update}(z_1, \vartheta^+, \vartheta^-, z_2)$, by which an incomplete state z_1 is updated to z_2 according to positive and negative effects ϑ^+ and ϑ^-, resp. The first two clauses in Fig. 4 encode macro (1). Correctness of this definition

```
holds(F,[F|_]).
holds(F,Z) :- nonvar(Z), Z=[F1|Z1], \+ F==F1, holds(F,Z1).

holds(F,[F|Z],Z).
holds(F,Z,[F1|Zp]) :- nonvar(Z), Z=[F1|Z1], \+ F==F1, holds(F,Z1,Zp).

minus(Z,[],Z).
minus(Z,[F|Fs],Zp) :- ( \+ not_holds(F,Z) -> holds(F,Z,Z1) ;
                        \+ holds(F,Z)     -> Z1 = Z ;
                        cancel(F,Z,Z1), not_holds(F,Z1) ),
                      minus(Z1,Fs,Zp).
plus(Z,[],Z).
plus(Z,[F|Fs],Zp) :- ( \+ holds(F,Z)     -> Z1=[F|Z] ;
                       \+ not_holds(F,Z) -> Z1=Z ;
                       cancel(F,Z,Z2), Z1=[F|Z2], not_holds(F,Z2) ),
                     plus(Z1,Fs,Zp).

update(Z1,ThetaP,ThetaN,Z2) :- minus(Z1,ThetaN,Z), plus(Z,ThetaP,Z2).
```

Fig. 4. The foundational clauses for reasoning about actions

follows from the foundational axioms of decomposition and irreducibility. The ternary $Holds(f, z, z')$ encodes $Holds(f, z) \wedge z' = z - f$. The following proposition implies that the definition is correct wrt. the macro definition of fluent removal, under the assumption that lists are free of duplicates.

Proposition 3. *Axioms* $\Sigma_{state} \cup \{z = f_1 \circ z_1 \wedge \neg Holds(f_1, z_1)\}$ *entails*

$$\begin{aligned} &Holds(f, z) \wedge z' = z - f \equiv \\ &\quad f = f_1 \wedge z' = z_1 \\ &\quad \vee \ (\exists z'')\,(f \not= f_1 \wedge Holds(f, z_1) \wedge z'' = z_1 - f \wedge z' = f_1 \circ z'') \end{aligned}$$

Proof. Suppose $f = f_1$. If $Holds(f, z) \wedge z' = z - f$, then $z' = (f_1 \circ z_1) - f_1$ since $z = f_1 \circ z_1$; hence, $z' = z_1$ since $\neg Holds(f_1, z_1)$. Conversely, if $z' = z_1$, then $z' = (f_1 \circ z_1) - f_1 = z - f$, and $Holds(f, z)$ since $z = f_1 \circ z_1$ and $f_1 = f$.

Suppose $f \not= f_1$. If $Holds(f, z)$ and $z' = z - f$, then $Holds(f, z_1)$ and $z' = (f_1 \circ z_1) - f$; hence, there is some z'' such that $z'' = z_1 - f$ and $z' = f_1 \circ z''$. Conversely, if $Holds(f, z_1) \wedge z'' = z_1 - f \wedge z' = f_1 \circ z''$, then $Holds(f, z)$ and $z' = (f_1 \circ z_1) - f$; hence, $Holds(f, z) \wedge z' = z - f$.

Removal and addition of finitely many fluents is defined recursively. The recursive clause for `minus` says that if $\neg Holds(f, z)$ is unsatisfiable (that is, f is known to hold in z), then subtraction of f is given by the definition of the ternary $Holds$ predicate. Otherwise, if $Holds(f, z)$ is unsatisfiable (that is, f is known to be false in z), then hence $z - f$ equals z. If, however, the status of the fluent is not entailed by the state specification at hand for z, then partial knowledge of f in $\Phi(z)$ may not transfer to the resulting state $z - f$ and, hence,

```
cancel(F,Z1,Z2) :- var(Z1) -> cancel(F,Z1), cancelled(F,Z1), Z2=Z1 ;
                   Z1=[G|Z], ( F\=G -> cancel(F,Z,Z3), Z2=[G|Z3]
                                     ; cancel(F,Z,Z2) ).

cancel(F,Z)~not_holds(G, Z)     <=>              \+ F\=G | true.
cancel(F,Z)~not_holds_all(G, Z) <=>              \+ F\=G | true.
cancel(F,Z)~or(V,Z)             <=> member(G,V), \+ F\=G | true.

cancel(F,Z), cancelled(F,Z) <=> true.
```

Fig. 5. Auxiliary clauses and CHRs for canceling partial information about a fluent

needs to be cancelled. Consider, for example, the partial state specification

$$Holds(F(y), z) \ \wedge \ [\, Holds(F(A), z) \vee Holds(F(B), z) \,] \tag{19}$$

This formula does not entail $Holds(F(A), z)$ nor $\neg Holds(F(A), z)$. So what can be inferred about the state $z - F(A)$? Macro expansion of "$-$" implies that $\Sigma_{state} \cup \{(19)\} \cup \{z_1 = z - F(A)\}$ entails $\neg Holds(F(A), z_1)$. But it does not follow whether $F(y)$ holds in z_1 or whether $F(B)$ does, since

$$\begin{aligned} \Sigma_{state} \cup \{(19)\} \cup \{z_1 = z - F(A)\} \models \\ [\, y = A \supset \neg Holds(F(y), z_1) \,] \wedge \\ [\, y \not= A \supset Holds(F(y), z_1) \,] \wedge \\ [\, \neg Holds(F(B), z) \supset \neg Holds(F(B), z_1) \,] \wedge \\ [\, Holds(F(B), z) \supset Holds(F(B), z_1) \,] \end{aligned}$$

Therefore, in the clause for `miunus` all partial information concerning f in the current state z is cancelled prior to asserting that f does not hold in the resulting state. The definition of cancellation of a fluent f is given in Fig. 5. In the base case, all negative and disjunctive state information affected by f is cancelled via the constraint `cancel`(f, z). The latter is resolved itself by the auxiliary constraint `cancelled`(f, z), indicating the termination of the cancellation procedure. In the recursive clause for `cancel`(f, z_1, z_2), each atomic, positive state information that unifies with f is cancelled.

In a similar fashion, the recursive clause for `plus` says that if $Holds(f, z)$ is unsatisfiable (that is, f is known to be false in z), then f is added to z; otherwise, if $\neg Holds(f, z)$ is unsatisfiable (that is, f is known to hold in z), then $z + f$ equals z. But if the status of the fluent is not entailed by the state specification at hand for z, then all partial information about f in z is cancelled prior to adding f to the state and asserting that f does not hold in the tail.

The definitions for `minus` and `plus` imply that a fluent to be removed or added does not hold or hold, resp., in the resulting state. Moreover, by definition cancellation does not affect the parts of the state specification which do not unify

with the fluent in question. Hence, these parts continue to hold in the resulting state after the update. The correctness of this encoding of update follows from the main theorem of the Fluent Calculus, which says that the axiomatization of state update by the macros for "−" and "+" solves the frame problem [11]: A fluent holds in the updated state just in case it either holds in the original state and is not subtracted, or it is added.

In the accompanying paper [12], it is shown how this CLP-based approach to reasoning about actions can be used as the kernel for a high-level programming method which allows to design cognitive agents that reason about their actions and plan. Thereby, agents use the concept of a state as their mental model of the world when conditioning their own behavior or when planning ahead some of their actions with a specific goal in mind. As they move along, agents constantly update their world model in order to reflect the changes they have effected. This maintaining the internal state is based on the definition of so-called state update axioms for each action, which in turn appeal to the definition of update as developed in Section 4. Thanks to the extensive reasoning facilities provided by the kernel of FLUX and in particular the constraint solver, the language allows to implement complex strategies with concise and modular agent programs.

5 Computational Behavior

Experiments have shown that FLUX scales up well. In the accompanying paper [13], we report on results with a special variant of FLUX for complete states applied to a robot control program for a combinatorial mail delivery problem. The experiments show that FLUX can compute the effects of hundreds of actions per second. The computational behavior of FLUX and the constraint solver in the presence of incomplete states has been analyzed with an agent program for the office cleaning domain, by which the robot systematically explores its partially known environment and acts cautiously under incomplete information. The results show that there is but a linear increase in the action computation cost as the knowledge of the environment grows. Notably, due to the state-based paradigm, action selection and update computation never depends on the history of actions. Therefore, FLUX scales up effortlessly to arbitrarily long sequences of actions. This result has been compared to GOLOG [6], where the curve for the computation cost suggests a polynomial increase over time [13].

6 Summary

We have presented a CLP-based approach to reasoning about actions in the presence of incomplete states based on Constraint Handling Rules and finite domain constraints. Both the constraint solver and the logic program for state update have been verified against the action theory of the Fluent Calculus. The experiments reported in [13] have shown that the constraint solver scales up well. This is particularly remarkable since the agent needs to constantly perform theorem proving tasks when conditioning its behavior on what it knows about

the environment. Linear performance has been achieved due to a careful design of the state constraints supported in our approach; the restricted expressiveness makes theorem proving computationally feasible. Future work will be to gradually extend the language, e.g., by constraints expressing exclusive disjunction, without loosing the computational merits of the approach.

References

1. Wolfgang Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986. 80
2. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998. 71, 76
3. Steffen Hölldobler and Josef Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990. 80
4. Robert Kowalski and M. Sergot. A logic based calculus of events. *New Generation Computing*, 4:67–95, 1986. 71
5. Yves Lespérance, Hector J. Levesque, Fangzhen Lin, D. Marcu, Ray Reiter, and Richard B. Scherl. A logical approach to high-level robot programming—a progress report. In B. Kuipers, editor, *Control of the Physical World by Intelligent Agents, Papers from the AAAI Fall Symposium*, pages 109–119, New Orleans, LA, November 1994. 70
6. 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. 71, 83
7. John McCarthy. *Situations and Actions and Causal Laws*. Stanford Artificial Intelligence Project, Memo 2, Stanford University, CA, 1963. 71
8. Raymond Reiter. *Logic in Action*. MIT Press, 2001. 71
9. Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997. 71
10. Murray Shanahan and Mark Witkowski. High-level robot control through logic. In C. Castelfranchi and Y. Lespérance, editors, *Proceedings of the International Workshop on Agent Theories Architectures and Languages (ATAL)*, volume 1986 of *LNCS*, pages 104–121, Boston, MA, July 2000. Springer. 71
11. 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. 71, 72, 80, 83
12. Michael Thielscher. Programming of reasoning and planning agents with FLUX. In D. Fensel, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 435–446, Toulouse, France, April 2002. Morgan Kaufmann. 71, 83
13. Michael Thielscher. Pushing the envelope: Programming reasoning agents. In Chitta Baral and Sheila McIlraith, editors, *AAAI Workshop on Cognitive Robotics*, Edmonton, Canada, July 2002. AAAI Press. 71, 83

Using Hybrid Concurrent Constraint Programming to Model Dynamic Biological Systems

Alexander Bockmayr and Arnaud Courtois

Université Henri Poincaré, LORIA
B.P. 239, F-54506 Vandœuvre-lès-Nancy, France
{bockmayr,acourtoi}@loria.fr

Abstract. Systems biology is a new area in biology that aims at achieving a systems-level understanding of biological systems. While current genome projects provide a huge amount of data on genes or proteins, lots of research is still necessary to understand how the different parts of a biological system interact in order to perform complex biological functions. Computational models that help to analyze, explain or predict the behavior of biological systems play a crucial role in systems biology. The goal of this paper is to show that hybrid concurrent constraint programming [11] may be a promising alternative to existing modeling approaches in systems biology. Hybrid cc is a declarative compositional programming language with a well-defined semantics. It allows one to model and simulate the dynamics of hybrid systems, which exhibit both discrete and continuous change. We show that Hybrid cc can be used naturally to model a variety of biological phenomena, such as reaching thresholds, kinetics, gene interaction or biological pathways.

1 Introduction

The last decades have seen a tremendous progress in molecular biology, the most spectacular result being the announcement of a first draft of the entire human genome sequence in June 2000, with analyses published in February 2001. Current genome, transcriptome or proteome projects, whose goal is to determine completely all the genes, RNA or proteins in a given organism, produce an exponentially growing amount of data. Storing, maintaining, and accessing these data represents already a challenge to computer science. But the real work - with an enormous impact on medicine and pharmacy - consists in exploiting all these data and in understanding how the various components of a biological system (i.e. genes, RNA, proteins etc.) interact in order to perform complex biological functions.

Systems biology is a new area in biology, which aims at a system-level understanding of biological systems [16]. While traditional biology examines single genes or proteins in isolation, system biology simultaneously studies the complex interaction of many levels of biological information - genomic DNA, mRNA,

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 85–99, 2002.
© Springer-Verlag Berlin Heidelberg 2002

proteins, informational pathways and networks - to understand how they work together, see [14] for a recent example.

The development of computational models of biological systems plays a crucial role in systems biology [3]. A number of projects like BioSpice, Cellerator, DBSolve, E-Cell, Gepasi, Jarnac, ProMot/DIVA, StochSim or Virtual Cell aim at modeling and simulating biological processes. The Systems Biology Workbench [13] is a software platform currently being developed in order to enable the different tools to interact with each other. From a programming language perspective, a *fundamental question* arises: what is the semantics underlying these different approaches and the possible combinations between them?

The goal of this paper is to present hybrid concurrent constraint programming (`Hybrid cc`) [11] as a promising alternative to existing modeling and simulation approaches in systems biology. `Hybrid cc` is a very powerful framework for modeling, analyzing and simulating *hybrid systems*, i.e., systems that exhibit both discrete and continuous change. It is a declarative compositional programming language based on the `cc` paradigm. From a computer science perspective, a major advantage of `Hybrid cc` compared to other approaches is that it is a full programming language with a well-defined semantics, based on a small number of primitives. From the viewpoint of systems biology, these basic constructs may help to identify key computational concepts needed to represent and to understand biological systems at the molecular and cellular level.

The organization of this paper is as follows. We start in Sect. 2 by giving a short overview of modeling approaches for molecular and cell biology. We emphasize the role of hybrid systems, which can cover both discrete and continuous phenomena. Sect. 3 recalls some of the basic ideas underlying hybrid concurrent constraint programming and gives a short introduction into the system `Hcc` that we used in our experiments. Sect. 4 is the core of the paper, explaining how `Hybrid cc` can be used to model biological systems in a high-level and declarative way. We show that various phenomena in biology, like thresholds, kinetics, gene interactions, or biological pathways have their natural counterpart in `Hybrid cc`. Sect. 5 summarizes the discussion and points out directions for further research. The results presented in this paper were first announced in [2], see also [5].

2 Existing Modeling Approaches

A variety of formalisms for modeling biological systems has been proposed in the literature. A detailed discussion goes far beyond the scope of this paper, we refer to [6,3] for an overview. Following [9], we may distinguish three basic approaches

- discrete,
- continuous,
- stochastic,

and various combinations between them.

Discrete models are based on discrete variables and discrete state changes. A classical example are Boolean networks for gene regulation [15]. For each gene, there is a Boolean variable indicating whether or not the gene is expressed in a given state. Boolean functions are then used to relate the variables belonging to different states. Qualitative networks [23] are an extension of Boolean networks, based on multivalued logic. Each variable now has a finite domain of possible values, which can be used, for example, to represent different levels of gene expression. Several authors have started to apply concepts and tools from formal verification to model biological systems, like Petri nets [18] or the π-calculus [19].

Continuous models have been used in mathematical biology for a very long time [25]. They are based on differential equations that typically model biochemical reactions. By making certain assumptions, any system of chemical reactions and physical constraints can be transformed into a system of nonlinear ordinary differential equations, whose variables are concentrations of proteins, RNA or other molecules [9].

In many models of biological systems, there exist discontinuous transitions. For example, we may want to model that expression of gene x activates expression of gene y; above a certain threshold, gene y inhibits expression of gene x. This leads to a system of conditional differential equations like

$$\begin{cases} \text{if } (y < 0.8) \text{ then } x' = -0.02 * x + 0.01 \\ \text{if } (y \geq 0.8) \text{ then } x' = -0.02 * x \\ y' = 0.01 * x \end{cases}$$

see Fig. 1 for an illustration.

The need to capture both discrete and continuous phenomena motivates the study of hybrid dynamical systems [24]. Their relevance for biology has been pointed out, among others, in [17] and [1], where special emphasis is put on the similarity between hybrid systems encountered in engineering and genetic circuits or biomolecular networks.

Another important issue for biology are stochastic phenomena. Probabilities occur in many different ways. We may distinguish between discrete and contin-

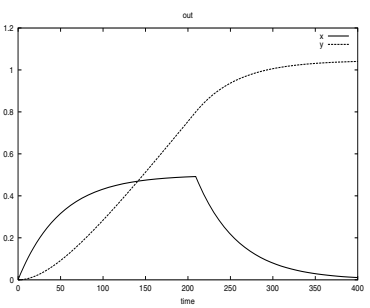


Fig. 1. Interaction between two genes

uous distributions. Probabilities may appear explicitly in random variables and random numbers, or implicitly like in kinetic laws or models like the one presented in Sect. 4.2. Typical situations where probabilities are used in modeling include:

- Simplified representation of processes (e.g. simulating the pathway choice of λ-phages in a population [8]).
- Integration of stochastic noise in order to get more realistic models.
- Large-scale Monte Carlo simulations.

3 Hybrid Concurrent Constraint Programming

Constraint programming started inside logic programming around 1985. In a constraint program, the user specifies a number of constraints. Each constraint defines a relation between variables that describe the state of the system under investigation. The constraint solver provides algorithms which compute solutions, i.e., valuations of the variables satisfying all the constraints, or which infer new constraints from the given ones.

In concurrent constraint programming (`cc`), different computation processes may run concurrently. Interaction is possible via the *constraint store*. The store contains all the constraints currently known about the system. A process may *tell* the store a new constraint, or *ask* the store whether some constraint is entailed by the information currently available, in which case further action is taken [20].

The original `cc` framework has been extended in various directions. In the context of this paper, we are interested in the following extensions:

- `Timed cc` [21]
- `Timed Default cc` [22]
- `Hybrid cc` [12,11,4]

One major difficulty in the original `cc` framework is that `cc` programs can detect only the presence of information, not its absence. To overcome this problem, [21] proposed to add to the `cc` paradigm a sequence of phases of execution. At each phase, a `cc` program is executed. At the end, absence of information is detected, and used in the next phase. This results in a synchronous reactive programming language, `Timed cc`. But, the question remained how to detect negative information instantaneously. `Default cc` extends `cc` by a negative ask combinator `if` a `else` A, which imposes the constraints of A unless the rest of the system imposes the constraint a. Logically, this can be seen as a default. Introducing phases as in `Timed cc` leads to `Timed Default cc` [22]. Only one additional construct is needed: `hence` A, which starts a copy of A in each phase after the current one.

`Hybrid cc` is an extension of `Default cc` over continuous time. First continuous constraint systems are allowed, i.e., constraints may involve differential equations that express initial value problems. Second, the `hence` operator is interpreted over continuous time. It imposes the constraints of A at every real time

Table 1. Combinators of `Hybrid cc`

Agents	Propositions
c	c holds now
`if` c `then` A	if c holds now, then A holds now
`if` c `else` A	if c will not hold now, then A holds now
`new` X `in` A	there is an instance $A[T/X]$ that holds now
A, B	both A and B hold now
`hence` A	A holds at every instant after now
`always` A	same as $(A,$ `hence` $A)$
`when(`c`)` A	same as `(if` c `then` A`, hence (if` c `then` A`))`
`unless(`c`)` A `else` B	same as `(if` c `then` B`, if` c `else` A`)`

instant after the current one. Tab. 1 summarizes the basic combinators of `Timed Default cc`. Note that in the implementation `Hcc` that we used, the syntax is slightly different.

The evolution of a system in `Hybrid cc` is piecewise continuous, with a sequence of alternating point and interval phases. All discrete changes take place in a point phase, where a simple `Default cc` program is executed. In a continuous phase, computation proceeds only through the evolution of time. The interval phase, whose duration is determined in the previous point phase, is exited as soon as the status of a conditional changes, see [12] for additional details.

`Hcc`, the current implementation of `Hybrid cc`, supports two types of constraints, which are handled by interval methods:

- ordinary differential equations, and
- (nonlinear) algebraic constraints.

Algebraic constraints are solved using a combination of interval propagation, splitting, Newton-Raphson method, and the Simplex algorithm. The ordinary differential equations are integrated using a 5th-order Runge-Kutta method with adaptive step size, modified for interval variables. For more details, we refer to [4], which also discusses the performance of the `Default cc` solver.

4 Modeling Biological Systems

4.1 Key Features of Biological Systems and Their Counterpart in Hybrid cc

The goal of this section is to show that `Hybrid cc` is well-suited for modeling and simulating dynamic biological systems. Tab. 2 gives an overview of a number of important features of biological systems and their counterpart in `Hybrid cc`.

Reaching thresholds. Thresholds can be used to determine when a process should be started or stopped. They arise in different situations. On the one hand, they may be introduced in the context of qualitative reasoning in order to discretize

Table 2. Biological systems in `Hybrid cc`

Biology	`Hybrid cc`
reaching thresholds	discrete events
time, concentration	continuous variables
kinetics	differential equations
gene interaction	concurrency
stochastic behavior	random numbers

continuous phenomena. On the other hand, they allow one to refine a model by specifying different dynamics for different states. A typical example for this is the multimode continuous model developed in Sect. 4.3. The kind of reactivity that can be expressed in `cc` languages depends on the class of the language: `Timed cc` allows for synchronous, `Default cc` for instantaneous, and `Hybrid cc` for asynchronous reactions.

Time, concentrations. Time and concentrations of molecules are continuous variables in a hybrid system. If the concentrations reach certain thresholds, this may trigger a change of the system state.

Kinetics. The kinetics of a biological system is represented by biochemical laws (differential equations) depending on continuous variables such as concentrations and real time.

Interactions. Interactions between different agents may be direct (e.g. molecule/gene) or indirect (e.g. gene/gene, molecule/effector/target). They reflect a concurrency at the biological level, which can be expressed naturally in a concurrent programming language.

Stochastic aspects. Probabilistic choice between different pathways or stochastic noise are common techniques used in modeling biological systems [3]. Probabilities may be realized by discrete probability distributions or in a hybrid way (e.g. the Langevin approach, where differential equations are extended by a noise term that induces stochastic fluctuations of the state about its deterministic value).

In the following sections, we present a number of models of dynamic biological systems that we have developed in `Hcc`. Although these examples are simple, they illustrate generic features arising also in much more complex models. Our goal is to show that `Hybrid cc` allows one to model biological systems in a natural and declarative way. Often it is possible to represent biological phenomena directly by corresponding statements in `Hybrid cc`.

4.2 Stochastic Behavior of Protein-DNA Complexation

Our first model represents the unstable binding mechanism between 2 kinds of proteins and a single DNA strand [9]. The system is composed of m M-proteins

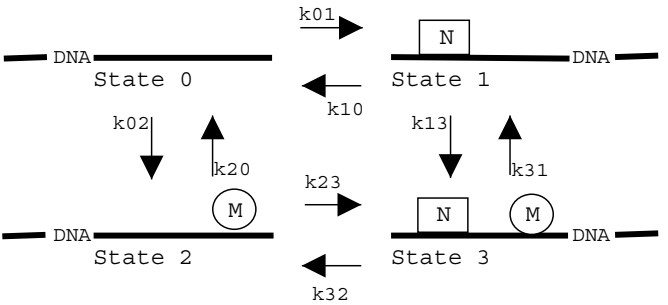


Fig. 2. Possible states and transitions

and n N-proteins. There are 4 possible states and 8 possible reactions, see Fig. 2. Each reaction is characterized by a stoichiometric coefficient. Note that "+" means a reaction, while "·" stands for a complexation :

$$N + DNA \underset{k_{10}}{\overset{k_{01}}{\rightleftharpoons}} N \cdot DNA \qquad N + M \cdot DNA \underset{k_{32}}{\overset{k_{23}}{\rightleftharpoons}} N \cdot M \cdot DNA$$
$$M + DNA \underset{k_{20}}{\overset{k_{02}}{\rightleftharpoons}} M \cdot DNA \qquad M + N \cdot DNA \underset{k_{31}}{\overset{k_{13}}{\rightleftharpoons}} M \cdot N \cdot DNA$$

We first consider a discrete time model. The probabilities $P_i(t+\Delta t)$ of being in state i at time $t+\Delta t$ depends on the probabilities at time t. They are computed in the following way:

$$\begin{pmatrix} P_0(t+\Delta t) \\ P_1(t+\Delta t) \\ P_2(t+\Delta t) \\ P_3(t+\Delta t) \end{pmatrix} = A \cdot \begin{pmatrix} P_0(t) \\ P_1(t) \\ P_2(t) \\ P_3(t) \end{pmatrix}$$

with the transition matrix

$$A = \begin{bmatrix} 1-nk_{01}\Delta t-mk_{02}\Delta t & k_{10}\Delta t & k_{20}\Delta t & 0 \\ nk_{01}\Delta t & 1-k_{10}\Delta t-mk_{13}\Delta t & 0 & k_{31}\Delta t \\ mk_{02}\Delta t & 0 & 1-k_{20}\Delta t-nk_{23}\Delta t & k_{32}\Delta t \\ 0 & mk_{13}\Delta t & nk_{23}\Delta t & 1-k_{31}\Delta t-k_{32}\Delta t \end{bmatrix}.$$

Note that there is no direct transition between the states 0 and 3 resp. 1 and 4. This discrete stochastic process can be expressed directly in `Hcc`. We use a clock for synchronization. The constant `dt` represents the time interval Δt, `prev(x)` yields the previous value of x. The constraints $P_i' = 0$ produce step functions for P_i, see Fig. 3a).

```
/* Define constants  */
#define dt  1         // interval length (delta t)
#define k01 0.02      //
```

```
...
/* Define variables */
interval clock; interval P0; ... ; interval P3;
/* Initialization (no binding) */
clock = dt; clock' = -0.5;
P0 = 1; P1 = 0; P2 = 0; P3 = 0;
/* Clock update */
always { if (clock=0) { clock' = .5;}
         else {clock'' = -1/dt;}
}
/* Compute probabilities */
always { if (clock=0) {
P0 = (1-n*k01*dt-m*k02*dt)*prev(P0) + k10*dt*prev(P1) + k20*dt*prev(P2);
P1 = n*k01*dt*prev(P0) + (1-k10*dt-m*dt*k13)*prev(P1) + k31*dt*prev(P3);
P2 = m*k02*dt*prev(P0) + (1-k20*d)-n*dt*k23)*prev(P2) + k32*dt*prev(P3);
P3 = m*k13*dt*prev(P1) + n*k23*dt*prev(P2) + (1-k31*dt-k32*dt)*prev(P3);
}
   else { P0' = 0; P1' = 0; P2' = 0; P3' = 0; }
}
/* Output */
sample(P0,P1,P2,P3);
```

This basic model can be extended in various ways. For example, we may assume that state 1 resp. 2 increase the production rate for some other proteins A and B. This may be modeled simply by adding constraints of the form

```
always {
    unless(P1>0.1)  {A' = 0.0001;}  else {A' = 0.03;};
    unless(P2>0.45) {B' = 0.00015;} else {B' = 0.01;};
}
```

resulting in the dynamics illustrated by Fig. 3b).

Another possibility is to consider $\Delta t \to 0$. This leads to a continuous model similar to the one developed in the next section.

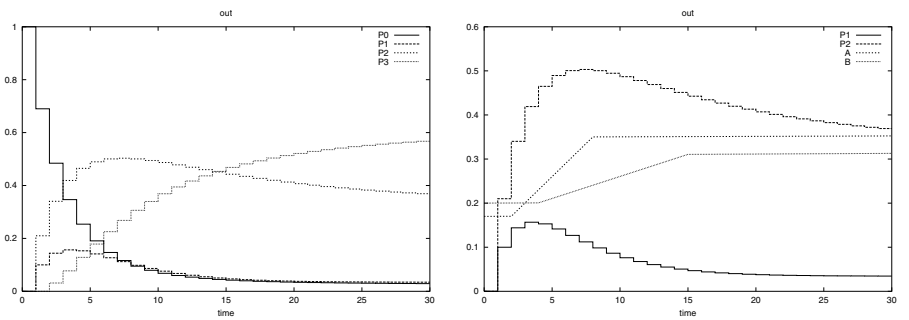


Fig. 3. a) Probabilities $P_0, \ldots, P_3$ b) P_i-dependent production of A and B

4.3 A Multimode Continuous Model

The kinetics of a chemical reaction depends on the concentration of the reactants. At very low and very high concentrations, the standard differential equations are no longer accurate. Therefore, in the following example, the system of differential equations is modified as soon as the concentration passes certain threshold values. This type of hybrid model can be seen as a refinement of a discrete qualitative network.

We model in `Hybrid cc` a phenomenon of bioluminescence for the bacteria V. fischeri [1]. These marine bacteria exist at low and high densities. While at low density the bacteria appear to be non-luminescent, a dramatic increase in luminescence can be observed when the density passes a given threshold. This phenomenon depends on the concentration of a certain small molecule A_i able to diffuse in and out of the cell membrane.

To describe the concentration of a molecular species x (RNA, protein, protein complex, or small molecule), we use the generic equation

$$\frac{dx}{dt} = v_s - v_d \pm v_r \pm v_t.$$

Here v_s is the synthesis rate, v_d the degradation rate, v_r the reaction rate w.r.t. other molecules, and v_t the transportation rate w.r.t. the environment (diffusion etc.).

We first present a mathematical model for one bacteria in the population [1]. The differential equations depend on the concentration of the molecule A_i. We use different equations depending on whether the concentration of A_i is low, medium, or high. Let x_7 resp. x_9 denote the internal and external concentration of A_i. The concentrations of the other molecules involved in the process are described by the variables $(x_1, x_2, x_3, x_4, x_5, x_6, x_8)$. All the remaining symbols are constants.

$$\frac{dx_1}{dt} = \begin{cases} \eta_1(\frac{c}{2} - x_1), & \text{if } 0 \le x_7 \le A_i^- \\ \frac{\eta_1}{4}(3c + \frac{x_8^{\nu_{81}}}{\kappa_{81}^{\nu_{81}} + x_8^{\nu_{81}}} - 4x_1), & \text{if } A_i^- < x_7 \le A_i^+ \\ -\eta_1 \cdot x_1, & \text{if } A_i^+ < x_7 \end{cases}$$

$$\frac{dx_2}{dt} = \begin{cases} -\eta_2 \cdot x_2, & \text{if } 0 \le x_7 \le A_i^- \\ \eta_2(3c + \frac{x_8^{\nu_{82}}}{\kappa_{82}^{\nu_{82}} + x_8^{\nu_{82}}} - x_2), & \text{if } A_i^- < x_7 \end{cases}$$

$$\vdots$$

$$\frac{dx_7}{dt} = -\eta_7 \cdot x_7 + r_4 \cdot x_4 - r_{mem}(x_7 - x_9) - r_{37,R} \cdot x_3 \cdot x_7$$

$$\frac{dx_9}{dt} = -\eta_7 \cdot x_9 + r_4 \cdot x_4 - r_{mem}(x_7 - x_9) + u$$

Again, the mathematical description can be directly translated into `Hcc`. Each molecular species is represented by an independent agent, whose dynamics is described by a single - possibly conditional - differential equation. The interaction between the different agents is controlled by the system.

```
#include "storeval.hcc"
/* Definition of constants */
   #define Ai_min 0.1 ... #define rmem 0.02
/* Definition of variables */
   interval x1; ... ; interval lum; interval pert;
/* Initialization */
   x1 = 0; ... ; lum = 0; storeval(pert,0);
/* Constraints and combinators */
   always { if (x7<Ai_min) x1' = mu1*((0.5*c)-x1);
    if ((x7>=Ai_min)&&(x7 < Ai_plus))
      x1' = (mu1/4)*((3*c)+((x8^ksi81)/((k81^ksi81)+(x8^ksi81)))-(4*x1));
    if (x7>=Ai_plus) x1' = -mu1*x1;}
   always { if (x7<Ai_min) x2' = -mu2*x2;
    if (x7>=Ai_min) x2'= mu2*(((x8^ksi82)/((k82^ksi82)+(x8^ksi82)))-x2);}
   ...
   always { x9' = (-mu7*x9)+(rmem*(x7-x9))+pert;}
/* Perturbation */
   when(time=200) storeval(pert,0.9);
   when(time=400) storeval(pert,0);
/* Luminescence  */
   always { lum' = x5'+x6';}
/* Output */
   sample(pert,x1,x2,x3,x4,x5,x6,x7,x8,x9,lum);
```

We assume that luminescence is proportional to x_5 and x_6. In order to simulate a population growth, we perform an artificial perturbation of the external concentration x_9 in the time interval $[200, 400]$. This is done by the function `storeval` that fixes a variable to a given value until the next change occurs. Fig. 4 shows the evolution of x_1 depending on x_7, as well as the change in luminescence caused by the virtual population growth in the interval $[200, 400]$. Note the change in the dynamics of x_1 when x_7 reaches the lower or upper threshold.

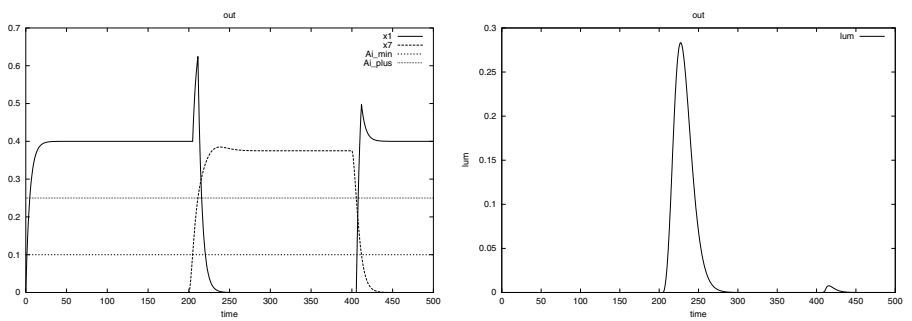


Fig. 4. Simulation of luminescence in V. Fischeri

4.4 A Typical Example of Hybrid Concurrent Constraint Programming

The next example combines in a single model all the features described earlier in Sec. 4.1. We consider cell differentiation for a population of epidermic *X. laevis* cells [7]. Each cell interacts with its neighbors, i.e., there are concurrent agents. The concentrations vary continuously. They are described by differential equations. Discrete transitions (see Fig. 5a) occur whenever certain threshold values are reached. Finally, the initialization is different for each cell (stochastic noise).

The cells are arranged in a hexagonal lattice (see Fig. 5b), so that a cell may have $2, 3, 4$ or 6 neighbors. In each cell, we consider the concentrations v_D, v_N of two proteins Delta and Notch. The Notch production capacity u_N in a cell depends on the Delta concentration of its neighbors (u_N is high if the ${v_D}^i$ are high):

$$u_N = \sum_{i=1}^{k} {v_D}^i, \text{ with } k \in \{2, 3, 4, 6\}.$$

The Delta production capacity u_D depends on the Notch concentration v_N in the same cell (u_D is high when v_N is low):

$$u_D = -v_N.$$

Depending on threshold values h_D, h_N, each cell can be in four different states:

State 1:	Delta and Notch inhibited:	$u_D < h_D$ and $u_N < h_N$
State 2:	Delta expressed, Notch inhibited:	$u_D \geq h_D$ and $u_N < h_N$
State 3:	Delta inhibited, Notch expressed:	$u_D < h_D$ and $u_N \geq h_N$
State 4:	Delta and Notch expressed:	$u_D \geq h_D$ and $u_N \geq h_N$

The production of Delta and Notch proteins depends on the state of the cell:

State 1:	$v'_D = -\lambda_D \cdot v_D,$	$v'_N = -\lambda_N \cdot v_N$
State 2:	$v'_D = R_D - \lambda_D \cdot v_D,$	$v'_N = -\lambda_N \cdot v_N$
State 3:	$v'_D = -\lambda_D \cdot v_D,$	$v'_N = R_N - \lambda_N \cdot v_N$
State 4:	$v'_D = R_D - \lambda_D \cdot v_D,$	$v'_N = R_N - \lambda_N \cdot v_N$

Here λ_D, λ_N are degradation rate constants, R_D, R_N are production rate constants.
Given the dimensions of the hexagonal lattice, we generate automatically the following `Hcc` program. To initialize the variables, we use a random number generator.

```
#include "parameters.hcc"
%module  "HCC_lib_math"
/* Declaration of variables */
   interval ud_1_1, un_1_1, vd_1_1, vn_1_1, state_1_1, ud_2_1, ...
/* Auxiliary functions */
   interval coeff_rd(interval ud) {
       if (ud<hd) return = 0;
```

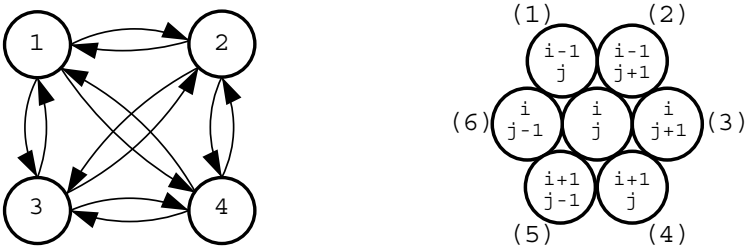


Fig. 5. a) States and transitions b) Hexagonal lattice

```
       else return = 1; }
   interval coeff_rn(interval un) {
       if (un<hn) return = 0;
       else return = 1; }
/* Initialization (with stochastic noise) */
   vd_1_1 = HCC_noise(1,30) ; vn_1_1 = HCC_noise(1,30) ; ...
/* Constraints and Combinators */
   always {
       un_1_1  =  vd_1_2  + vd_2_1; un_1_2  = ...
       ud_1_1  = -vn_1_1;           ud_1_2  = ...
       vd_1_1' = (coeff_rd(ud_1_1)*rd) - lambda_d * vd_1_1; vd_1_2' = ...
       vn_1_1' = (coeff_rn(un_1_1)*rn) - lambda_n * vn_1_1; vn_1_2' = ...
       if ((ud_1_1<hd) && (un_1_1<hn)) state_1_1 = 1; ...
   }
/* Output */
sample(state_1_1, state_1_2, state_2_1, ...);
```

The indices of the variables correspond to Fig. 5. The intracellular concentrations of Delta and Notch are initialized with a stochastic noise of $\pm 30\%$ about the value 1. Fig. 6a) illustrates a transition that modifies the concentration of Delta in a border cell of a 15x15 network. Fig. 6b) shows the steady state reached at the end of the simulation. Each cell is either in State 2, marked by `*`, or State 3, marked by `o` ("salt-and-pepper" effect). Thus, in spite of its simplicity, this hybrid model provides very accurate qualitative results.

4.5 Benefits of Hybrid cc for Modeling Biological Systems

In addition to the advantages of modeling in a language of the class `Timed Default cc` [22], the following features of `Hybrid cc` seem particularly useful in the context of biology.

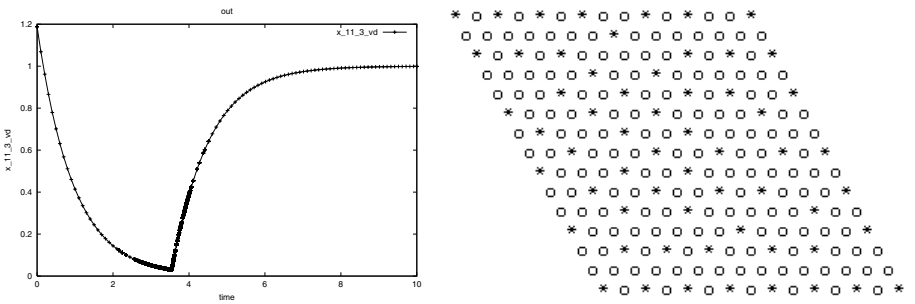


Fig. 6. a) Transition in a border cell b) Steady state

Features of Hybrid cc	Interest for biology
Concurrent agents	Clear separation between different processes. Possibility to refine the model.
Declarative programming	Only the constraints have to be stated. Control done implicitly by `Hybrid cc`.
Constraints	Express possibly incomplete knowledge. Define the granularity of the analysis.
Differential equations	Describe biochemical reactions and continuous variations.
Modularity	Isolate the core of a model, crucial for highly context-dependent models. Reusability.
Asynchronous reactions	Accurate for reactive systems.

5 Conclusion and Further Research

`Hybrid cc` is a promising alternative to existing modeling and simulation approaches in systems biology. From the programming language perspective, it is a highly expressive, compositional language based on the constraint programming paradigm, with a well-defined semantics. From the application point of view, `Hybrid cc` allows the biologist to model biological systems in a high-level and declarative way, using a small number of language constructs that capture directly a variety of biological phenomena. When needed, all the advantages of a full programming language are available.

There are various directions to continue this research. The examples presented in this paper were taken from the literature in order to illustrate the interest of `Hybrid cc` for modeling biological systems. They are relatively small. We have started recently a project with the Institut Pasteur in Paris in order to develop with `Hcc` a rather complex model of some aspects of the mammalian cell cycle. We hope that this collaboration will not only help to get a better understanding of the biological phenomena, but that it will also provide additional

insight into `Hybrid cc`. For example, it may turn out that the constraint solving capabilities of `Hybrid cc` have to be enhanced or that the language has to be extended in order to capture specific biologic phenomena. In particular, we may want to consider stochastic extensions like `Probabilistic cc` [10].

References

1. R. Alur, C. Belta, F. Ivancic, V. Kumar, M. Mintz, G. J. Pappas, H. Rubin, and J. Schug. Hybrid modeling and simulation of biomolecular networks. In *Hybrid Systems: Computation and Control, HSCC 2001*, pages 19–32. Springer, LNCS 2034, 2001. 87, 93
2. A. Bockmayr and A. Courtois. Modeling biological systems in hybrid concurrent constraint programming (Abstract). In *2nd Int. Conf. Systems Biology, ICSB'01, Pasadena, CA*, 2001. 86
3. J. M. Bower and H. Bolouri, editors. *Computational modeling of genetic and biochemical networks*. MIT Press, 2001. 86, 90, 98
4. B. Carlson and V. Gupta. Hybrid cc and interval constraints. In *Hybrid Systems: Computation and Control, HSCC'98*, pages 80 – 95. Springer, LNCS 1386, 1998. 88, 89
5. A. Courtois. Modélisation de systèmes biologiques en programmation par contraintes. Rapport de DEA (in French), Univ. Henri Poincaré, LORIA, July 2001. 86
6. H. de Jong. Modeling and simulation of genetic regulatory systems: a literature review. *Journal of Computational Biology*, 9(1):69 – 105, 2001. 86
7. R. Ghosh and C. Tomlin. Lateral inhibition through Delta-Notch signaling: A piecewise affine hybrid model. In *Hybrid Systems: Computation and Control, HSCC 2001*, pages 232–246. Springer, LNCS 2034, 2001. 95
8. M. A. Gibson and J. Bruck. A probabilistic model of a prokaryotic gene and its regulation. In Bower and Bolouri [3], chapter 2, pages 49–71. 88
9. M. A. Gibson and E. Mjolsness. Modeling the activity of single genes. In Bower and Bolouri [3], chapter 1, pages 1–48. 86, 87, 90
10. V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *26th ACM Conf. Principles of Programming Languages, POPL'99, San Antonio, CA*, pages 189 – 202. ACM, 1999. 98
11. V. Gupta, R. Jagadeesan, and V. Saraswat. Computing with continuous change. *Science of computer programming*, 30(1-2):3–49, 1998. 85, 86, 88
12. V. Gupta, R. Jagadeesan, V. Saraswat, and D. G. Bobrow. Programming in hybrid constraint languages. In *Hybrid Systems II*, pages 226–251. Springer, LNCS 999, 1995. 88, 89
13. M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. Doyle, and H. Kitano. The ERATO systems biology workbench: Enabling interaction and exchange between software tools for computational biology. *Pacific Symposium on Biocomputing*, 7, 2002. 86
14. T. Ideker, V. Thorsson, J. A. Ranish, R. Christmas, J. Buhler, J. K. Eng, R. Bumgarner, D. R. Goodlett, R. Aebersold, and L. Hood. Integrated genomic and proteomic analyses of a systematically perturbed metabolic network. *Science*, 292:929 – 934, May 2001. 86
15. S. A. Kauffman. *The origins of order*. Oxford Univ. Press, 1993. See also J. Theor. Biol. 22: 437, 1969. 87

16. H. Kitano, editor. *Foundations of system biology.* MIT Press, 2001. 85
17. H. Matsuno, A. Doi, M. Nagasaki, and S. Miyano. Hybrid petri net representation of gene regulatory network. *Pacific Symposium on Biocomputing*, 5:338–349, 2000. 87
18. V. N. Reddy, M. L. Mavrovouniotis, and M. N. Liebman. Petri net representation in metabolic pathways. In *Intelligent Systems for Molecular Biology, ISMB'93*, pages 328–336. AAAI Press, 1993. 87
19. A. Regev, W. Silverman, and E. Shapiro. Representation and simulation of biochemical processes using the π-calculus process algebra. *Pacific Symposium on Biocomputing*, 6:459–470, 2001. 87
20. V. A. Saraswat. *Concurrent constraint programming.* ACM Doctoral Dissertation Awards. MIT Press, 1993. 88
21. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *9th Symp. Logic in Computer Science, LICS'94, Paris*, pages 71 – 80. IEEE, 1994. 88
22. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5/6):475–520, 1996. 88, 96
23. D. Thieffry and R. Thomas. Qualitative analysis of gene networks. *Pacific Symposium on Biocomputing*, 3:77–88, 1998. 87
24. A. van der Schaft and H. Schumacher. *An introduction to hybrid dynamical systems.* Springer, Lecture Notes in Control and Information Sciences, Vol. 251, 2000. 87
25. E. O. Voit. *Computational analysis of biochemical systems.* Cambridge Univ. Press, 2000. 87

Efficient Real-Time Model Checking Using Tabled Logic Programming and Constraints*

Giridhar Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, New York, U.S.A.
{giri,cram,ram}@cs.sunysb.edu

Abstract. Logic programming based tools for real-time model checking are beginning to emerge. In a previous work we had demonstrated the feasibility of building such a model checker by combining constraint processing and tabulation. But efficiency and practicality of such a model checker were not adequately addressed. In this paper we describe XMC/dbm, an efficient model checker for real-time systems using tabling. Performance gains in XMC/dbmdirectly arise from the use of a lightweight constraint solver combined with tabling. Specifically the timing constraints are represented by difference bound matrices which are encoded as Prolog terms. Operations on these matrices, the dominant component in real-time model checking, are shared using tabling. We provide experimental evidence that the performance of XMC/dbmis considerably better than our previous real-time model checker and is highly competitive to other well known real-time model checkers implemented in C/C++. An important aspect of XMC/dbm is that it can handle verification of systems consisting of untimed components with performance comparable to verification systems built specifically for untimed systems.

1 Introduction

In the recent past several techniques for verification based on logic programming have been developed. For example, constraint logic programming has been used for the analysis and verification of hybrid and real-time systems [23,11]. Partial deduction techniques as well as CLP have been used for model checking infinite-state systems [7,6,17,18]. New logic program transformations have been devised for verifying parameterized systems [21,10] and more recent work includes verification of security protocols [5].

Tools based on these techniques, such as our XMC system are beginning to emerge. The XMC system is an efficient and flexible model checker for finite-state systems [19] implemented using the XSB tabled logic programming system [24]. The verification environment provided by the XMC system includes system and

* This work was supported in part by NSF grants EIA-9705998, CCR-9876242, and IIS-0072927.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 100–114, 2002.
© Springer-Verlag Berlin Heidelberg 2002

property specification languages, a simulator, model checker and a justifier for explaining the model checking results. The practicality of the XMC model checker has been demonstrated on several large-scale verification problems [20].

XMC is a model checker for untimed systems. But many reactive systems operate under real-time requirements. Model checking such systems requires constraint processing machinery, an observation that was the basis for several CLP based techniques [11,7,6,18]. In [9] we developed XMC/RT for model checking real-time systems using CLP with tabulation. It was implemented by adding a generic constraint solver for linear constraints over the reals to XSB, thereby deriving a constraint logic programming system with tabulation.

XMC/RT demonstrated the feasibility of building model checkers using logic programming technologies for real-time systems that can be competitive with those implemented in other languages such as C/C++. However to make it as usable a system as XMC, there were performance issues that remained to be addressed. Most notably, constraint handling in XMC/RT was not tightly integrated to the XSB system. In particular it used the POLINE polyhedra package [12], a generic constraint solver library routines (written in C++) for linear arithmetic constraints over reals. There are several drawbacks with this approach. First of all, this package had to be interfaced such that the constraints are stored and manipulated by the constraint solver, but handles to these constraints may be present on the Prolog side. This made any low-overhead management of storage for the the combined system difficult. Secondly, constraints over linear arithmetic are more general than those that arise in real-time model checking. The more general constraint solver imposes additional overheads. Lastly, delegating all the constraint processing to an external handler (such as POLINE) makes it difficult, if not impossible, to do low-level optimizations on the constraint operations. A tightly integrated light-weight constraint solver would facilitate such optimizations. It will significantly improve the time as well as memory utilization of a real-time model checker and thereby enhance its scalability to handle large verification problems. Moreover a beneficial fallout of a tight integration is that it provides a unified environment for model checking systems consisting of both timed as well as untimed components.

In this paper we describe the design and implementation of XMC/dbm, a real-time model checker that eliminates some of the problems in XMC/RT. The main idea is to use a representation for constraints that can be efficiently encoded and manipulated as Prolog terms. In XMC/dbm the timing constraints are represented using *Difference Bound Matrices (DBMs)* [8], an idea gainfully explored in Uppaal [16]. Constraints on clocks can be concisely represented in DBMs and efficiently manipulated. Most of the expensive computation in real-time model checking centers around manipulating such constraints. Hence sharing such computations via caching becomes critical for performance. A tabled logic programming system offers a natural platform for implementing such a model checker. Specifically, in XMC/dbm the computations over DBMs are tabled. Combining this with other generic as well as Prolog-specific optimizations we obtain considerable performance gains in time and space. XMC/dbm's performance is

considerably better than XMC/RT and is competitive with other well known real-time model checkers such as Uppaal [15] and HyTech [13].

Traditionally timed and untimed systems have been viewed as distinct verification domains. An important aspect of XMC/dbm (inherited from XMC/RT) is that it can handle verification of systems consisting of timed as well as untimed components. It is accomplished without unduly compromising the performance of model checking untimed systems. Finally it is interesting to note that so far the superior termination properties of a tabled logic programming system have been exploited for implementing model checkers. XMC/dbm demonstrates that the naturalness of caching that "comes for free" in such systems can also play an equally important role.

In terms of related work, several researchers, as noted earlier, have also used (constraint) logic programming for the verification of real-time and other infinite-state systems. The distinguishing aspect of the current work is that instead of exploiting the expressiveness of constraint logic programming to expand the set of systems that can be modeled, our primary emphasis is on building an efficient, usable and unified system that scales to real-world verification problems.

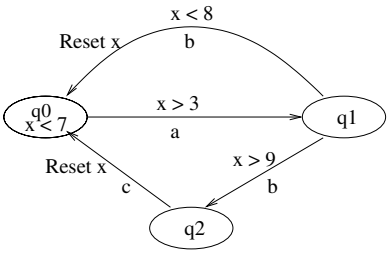
The rest of this paper is organized as follows. In Section 2.1 we present a brief discussion on timed automata, timed modal mu-calculus and model checking of timed automata with timed modal mu-calculus. The encoding of DBMs and related operations as logic programs, as well as optimizations on these operations are described in Section 3. We present experimental evaluation of XMC/dbm in Section 4 and the impact of optimizations used in the implementation in Section 5. Concluding remarks appear in Section 6.

2 Preliminaries

2.1 Timed Automata

Timed safety automata (TSA) [1] model real-time systems by augmenting finite-state automata with real-valued clocks. A TSA has a finite set of control *locations* (corresponding to the states in a finite-state automaton) and a set of *transitions* between these locations, and a finite set of *clocks* associated with the automaton. A *state* of an automaton is characterized by a control location (the "current" location) and a valuation of clocks. A transition may be annotated with an *action label* (drawn from a finite set of symbols) which is used for synchronization between multiple TSAs; a *guard* which is a constraint over clock valuations that must be satisfied for that transition to be enabled; and a set of clocks that are *reset* to zero when that transition is taken. Each location is associated with a location *invariant*, a constraint on clock valuations which must be satisfied when an automaton remains at that location. Constraints on clocks (location invariants and guards) in a TSA are a conjunction of base constraints of the form $x \odot k$ and $x - y \odot k$, where x and y are clocks, $\odot$ is one of $\{<, \leq, >, \geq\}$ and k is an integer constant.

Transitions are instantaneous: no time elapses when a transition is taken; however, a TSA may choose to remain at a location for an arbitrary period of



(a)

```
invariant(state(q0,X)) :-
   X < 7.
trans(state(q0,X), a, state(q1,X)) :-
   X > 3. % Guard
trans(state(q1,X0), b, state(q0,X1)) :-
   X0 < 8, X1 = 0. % Guard, Reset
trans(state(q1,X), b, state(q2,X)):-
   X > 9. % Guard
trans(state(q2,_), c, state(q0,X)) :-
   X = 0. % Reset
```

(b)

Fig. 1. An example Timed Safety Automaton (a) and its representation as a constraint logic program (b)

time as long as the location invariant is satisfied. A real-time system is modeled as a synchronous parallel composition of TSAs. All the clocks in a system progress at a constant rate. We assume that the real-time system is non-Zeno: the system does not remain in the same location indefinitely.

TSA can be represented as a constraint logic program by encoding the transition relation. For instance, the TSA given in Figure 1(a) is represented by the constraint logic program in Figure 1(b). Each state of the system is encoded by the term `state(`l`,` $\overline{C}$`)` where l denotes the location and $\overline{C}$ denotes zero or more clocks. Transitions without guards or resets are represented simply as facts. For transitions without resets the clock valuations at the source and destination states remain unchanged.

TSAs can be augmented with discrete variables, which are non-clock, finite-domain variables; note that this does not increase the expressive power of TSAs. When two or more TSAs are composed in parallel, the locations of the composed system will be tuples, each element of the tuple denoting the location of the corresponding component TSA. The clocks of the composed TSA will be the union of the set of clocks in each component TSA. It is useful to consider as a transition the evolution of a system where it stays in the same location while time elapses. We call such special transitions as *delay* transitions.

The constraint-based representation of TSAs permit us to talk uniformly about a single valuation of clocks (where each clock is mapped to a unique real number) as well as a set of valuations that show the same behavior. A set of clock valuations that can be represented by as a conjunction of a finite number of clock constraints is called a *zone*. Note that by definition, a zone is a convex region of the clock valuation space.

2.2 Timed Modal Mu-Calculus

Timed modal mu-calculus [22] adds time modalities $\langle\epsilon\rangle$ and $[\epsilon]$ and resettable clocks to the modal mu-calculus. We use the following syntax for timed modal mu-calculus formulas (from [9]):

```
F -->  tt | ff | atomic(C)
     | and(F, F) | or(F, F) | neg(F)
     | diam(A, F) | box(A, F)
     | epsdiam(F) | epsbox(F) | reset(Z, F)
     | form(X)
```

where `tt` and `ff` denote constants *true* and *false* respectively; `atomic(C)` is a base formula where `C` is an atomic proposition or a constraint over system and formula clocks; `and`, `or` and `not` are standard logical connectives; `diam(A, F)` and `box(A, F)` are dual modal operators from classical modal mu-calculus (`diam(A,F)` asserts that there exists a transition labeled `A` after which `F` holds; `box(A,F)` asserts that `F` holds after every transition labeled `A`); `epsdiam(F)` (after some delay `F` holds) and `epsbox(F)` (after every delay `F` holds) are dual time modalities; `reset(Z,F)` defines a new clock `Z` local to formula `F`; and `form(X)` refers to a mu-calculus variable `X` defined using fixed point equations of the form `X += F` (least fixed point) or `X -= F` (greatest fixed point).

For example, consider the TSA in Figure 1 in location `q1` with clock $X = 7$ (denoted by `state(q1, 7)`). A `b` transition is possible in this state, and hence the formula `diam(action(b), tt)` holds at `state(q1, 7)`. It follows that `epsdiam(diam(action(b), tt))` also holds at `state(q1, 7)` (i.e., with zero delay). Now consider `state(q1, 8)`. The formula `diam(action(b), tt)` no longer holds, but `epsdiam(diam(action(b), tt))` holds at `state(q1, 8)` since a `b` transition is possible after a delay of > 1 second. It also follows that `epsbox(diam(action(b), tt))` does not hold at `state(q1, 8)`.

The model of a timed modal mu-calculus formula is defined with respect to structures called dense labeled transition systems that can be derived from TSAs. The model of a formula is a set of states in the given structure where the formula holds, and is defined inductively based on the syntax of the formula. Following XMC and XMC/RT we encode these inductive rules as a tabled logic program to derive a model checker. In XMC, the states are represented as Herbrand terms. In case of real-time systems, we choose an appropriate constraint representation to denote a set of states. In XMC/RT, we chose to use the POLINE polyhedra package to represent and manipulate constraints. In this paper, we use Difference Bound Matrices (DBMs), which are themselves represented as Prolog terms, to denote sets of states, and construct an efficient solver for DBMs.

2.3 Model Checking the Timed Modal Mu-Calculus

We use the formulation of real-time model checking presented in [9] and recalled in Figure 2. In that model checker, we defined a predicate `models(`R`, `F`, `Rs`)` that, given a *zone* R finds the largest (finite) set of zones Rs that model F

```
models(SS, F, SR) :- union(R, models1(SS, F, R), SR).

models1(SS, neg(F), SR) :-   % negation (due to greatest fixed points)
        models(SS, F, NegSR), diff(SS, NegSR, SR).

models1(SS, box(Act, F), SR) :-  % universal transition modality
        split(SS, Act, LSS),
        member(S, LSS),
        findall(TS, trans(S, Act, TS), TSS),
        all_models(TSS, F, S, Act, SR).

all_models([], _, _, _, []).
all_models([SS0|Rest], F, S, Act, SR) :-
        models(SS0, F, SR0),
        inverse_trans(SR0, Act, S, SR1),
        all_models(Rest, F, S, Act, SR2),
        conjunction(SR1, SR2, SR).

models1(SS, epsbox(F), SR) :-    % universal time modality
        univ_elim(D, (    trans(SS, e(D), TS),
                          models(TS, F, TR),
                          inverse_trans(TR, e(D), SS, SRD)
                    ), SRD, SR).
```

Fig. 2. Encoding of the XMC/RT model checker (from [9])

and are contained in R. Note that, for a real-time system made up of a parallel composition of TSAs each zone is a tuple of locations (each element of the tuple denoting the location the corresponding TSA is in) and a conjunction of clock constraints. As explained in [9], this formulation differs significantly from the finite-state model checker in XMC [20] where the binary `models` predicate simply checks if a given system state is in the model of a formula. The first argument to `models/3` in the real-time checker represents a *set* of states, not all of which may model the given formula (the second argument). We could assume that when the goal succeeds, the first argument will be narrowed to a set of states that do model the formula. However, for eliminating the universal quantifier over time delays that is introduced by the universal time modality we need the (complete) set of *all* states that model a given formula. We accomplish this by aggregating such a set (the third argument) using a constraint operation `union`.

Apart from `union` we use two basic constraint operations to manipulate zones in the definition of `models/3`: `diff/3` which computes the difference between two constraints and `conjunction` that computes the intersection of two constraints; a derived constraint operation `univ_elim` which is used to eliminate a universally quantified delay variable, implemented by using difference and projection operations on constraints; and two operations on constraints based on transitions

of a timed automata: `split` which splits a given zone according to a transition label, and `inverse_trans` which finds the subset of a source zone that takes the automaton into a given target zone.

In addition to these operations used directly by the `models/3` predicate, the real-time model checker also uses a number of constraint operations to construct zones and compute global transitions from the given timed-automata specifications. Instead of directly using a constraint logic programming system, the formulation in [9] presents the model checker in terms of a tabled logic program with explicit references to constraint solving operations. This design decision was made due to two orthogonal reasons. First, tabled resolution is central to the high-level implementation of the model checker and there is (as yet) no single system that integrates constraint solving over reals with tabled resolution. Secondly, encoding delay transitions in a constraint language can introduce significant overheads due to introduction of the delay variable D and its subsequent elimination. In contrast, since all clocks move at the same rate, the effect of delay transitions on clock valuations can be realized by direct constraint manipulation. In this paper, we follow the overall design of [9] and describe a DBM-based representation and manipulation of clock constraints.

3 Difference Bounds Matrices

Timed safety automata restrict clock constraints to those of the form $x_1 - x_2 \sim c$ and $x_1 \sim c$, where x_1 and x_2 are clocks, c is an integer constant and $\sim$ is one of $\{<, \leq\}$. These constraints can be represented in a matrix form, called a *difference bound matrix* (DBM) [8] with the rows and columns labeled by clocks. We assume that clocks are named $x_1, x_2, \ldots$. Each element $M_{i,j}$ in a DBM is a pair $(b, \sim)$ representing the constraint $x_i - x_j \sim b$: i.e., an upper bound on the difference between x_i and x_j. To represent constraints of the form $x_i \sim c$, we use a special 0th "clock" x_0 whose value is fixed at 0; thus $x_i \sim c$, which is equivalent to $x_i - x_0 \sim c$ is represented by entry $M_{i,0}$ and constraint $c \sim x_j$, which is equivalent to $x_0 - x_j \sim -c$ is represented by entry $M_{0,j}$. When there is no upper bound on $x_i - x_j$, then the entry $M_{i,j}$ is set to $(\infty, <)$.

Consider a timed automata with two clocks x_1 and x_2 with constraints $x_1 \geq 6, x_1 < 9$. A DBM representing these constraints is give in Figure 3(a). In DBM 3(a), since $x_1 < 9$ and $0 \leq x_2$, we can infer that $x_1 - x_2 < 9$: i.e., the entry $M_{1,2}$ can be changed from $(\infty, <)$ to $(9, <)$, in effect tightening the upper bound on $x_1 - x_2$. This yields DBM in Figure 3(b). No further nontrivial inferences are possible in DBM 3(b). A DBM where each entry is the tightest possible constraint that can be inferred from that DBM is said to be in *canonical form*.

When a DBM is in canonical form, the effect of each entry has been have been fully propagated through the DBM. Hence entries in a canonical DBM can be manipulated independently. Consider adding constraints $3 \leq x_2 \leq 7$ to the DBM in Figure 3(b). The resulting DBM appears in Figure 4(a), and its canonical form is in Figure 4(b).

$$\begin{pmatrix} (0,\leq) & (-6,\leq) & (0,\leq) \\ (9,<) & (0,\leq) & (\infty,<) \\ (\infty,<) & (\infty,<) & (0,\leq) \end{pmatrix} \quad \begin{pmatrix} (0,\leq) & (-6,\leq) & (0,\leq) \\ (9,<) & (0,\leq) & (9,<) \\ (\infty,<) & (\infty,<) & (0,\leq) \end{pmatrix}$$

(a) (b)

Fig. 3. DBMs for constraints $6 \leq x_1 < 9$; DBM (b) is the canonical form of DBM (a)

$$\begin{pmatrix} (0,\leq) & (-6,\leq) & (-3,\leq) \\ (9,<) & (0,\leq) & (9,<) \\ (7,\leq) & (\infty,<) & (0,\leq) \end{pmatrix} \quad \begin{pmatrix} (0,\leq) & (-6,\leq) & (-3,\leq) \\ (9,<) & (0,\leq) & (6,<) \\ (7,\leq) & (1,\leq) & (0,\leq) \end{pmatrix} \quad \begin{pmatrix} (0,\leq) & (-6,\leq) & (-3,\leq) \\ (\infty,<) & (0,\leq) & (6,<) \\ (\infty,<) & (1,\leq) & (0,\leq) \end{pmatrix}$$

(a) (b) (c)

Fig. 4. (a) DBM for constraints $6 \leq x_1 < 9 \wedge 3 \leq x_2 \leq 7$; (b) is the canonical form of DBM (a); DBM (c) represents the clock values after an arbitrary passage of time

Canonical forms enable efficient implementation of several constraint operations such as delays. For instance, consider the set of clock valuations that can be reached after an arbitrary delay from a given zone. The passage of time eliminates the upper bounds on the clocks, but the relationship between any two clocks is preserved (since all clocks progress uniformly). If a DBM is in canonical form, this delay operation can be performed by simply eliminating the upper bounds on individual clocks (i.e., the 0-th column). An arbitrary passage of time from DBMs in Figures 4(a) and (b) results in the DBM is in Figure 4(c).

3.1 Constraint Operations with DBMs

We define the following constraint operations over DBMs:

- `add_constraints(Pi1, Constraints, Pi2)`: Given a DBM `Pi1`, generates a new DBM `Pi2` by adding further constraints on the clocks, specified by the list `Constraints`. This operation is implemented by iterating through each element of the DBM, and replacing a DBM entry if there is a tighter bound on that value given in `Constraints`. The operation neither expects, nor generates, DBMs in canonical form.
- `delay(Pi1, Pi2)`: Given a DBM `Pi1`, computes the DBM `Pi2` reachable from `Pi1` by incrementing all clocks by an arbitrary amount (arbitrary delay). This operation is implemented by setting the upper bounds on all clocks, specified in the 0-th column, to infinity. The argument `Pi1` must be in canonical form; the output, `Pi2`, will also in canonical form.
- `canonical_form(Pi1, Pi2)`: Canonical form of a DBM over n clocks can be found in $O(n^3)$ time by adapting an all-pairs shortest path algorithm.

We use a modified version of the Floyd-Warshall algorithm [4], reordering an inner loop to work efficiently over the list-of-lists representation of DBMs.

- `apply_resets(Pi1, Clocks, Pi2)`: Given a DBM `Pi1`, generates a new DBM `Pi2` obtained by resetting the values of clocks specified in the list `Clocks` to zero. This operation assumes that the given argument `Pi1` is in canonical form. Reset can be done by setting the 0-th column and 0-th row entries of the given clocks to 0, but the result may not be in canonical form. It is easy to "adjust" the result to its canonical form as follows. For each clock x_i reset by this operation, we copy the 0-th column of the DBM to the i-th column, and the 0-th row to the i-th row. The resulting DBM will be in canonical form [25].
- `check_invariants(Constraints, Pi)`: Check if `Pi` $\wedge$ `Constraints` is satisfiable. This operation exploits the fact that invariant constraints specify only upper bounds of individual clocks and have no constraints that relate different clock values. If `Pi` is in canonical form, these upper bound constraints can be checked by simply verifying that the lower bounds specified by the DBM of `Pi` is consistent with the given upper bounds.
- `conjunction(Pi1, Pi2, PL)`: Given two DBMs `Pi1` and `Pi2`, computes their conjunction `Pi3`. Conjunction of two DBMs M and M' is a DBM M'' such that $M''_{i,j}$ contains the tighter of the two bounds specified by $M_{i,j}$ and $M'_{i,j}$. Hence conjunction is analogous to matrix addition, and can be done in $O(n^2)$ time where n is the number of clocks represented in the DBM.
- `diff(Pi1, Pi2, PL)`: Given a DBM `Pi1` and a DBM `Pi2`, computes the set of DBMs `PL` containing points in `Pi1` that are not in `Pi2`.
- `union(V, Goal, PL)`: Given a goal `Goal` that contains a variable `V`, `PL` is the canonical representation of all states s such that `Goal`$[s/V]$;
- `univ_elim(D, Goal, ZD, PL)`: Given `ZD`, a disjunction of constraints over *Vars* $\cup$ {`D`} representing the set of all solutions to `Goal`, `PL` is a disjunction of constraints over *Vars* such that $\forall v \in \mathbf{R}$ (`ZD` $\wedge$ `D` $= v$) $\Leftrightarrow$ `PL`.

3.2 Using DBM Operations

The constraint operations described above are used in the `models/3` predicate, as well as to generate transitions from the specifications of timed safety automata. We assume that, given the transition relation of each TSA, `global_trans/5` gives the transition relation of the parallel composition of the TSAs. The global transition relation specifies the source and destination locations (each is a tuple with components representing local locations), the action label, guards and resets on the transition. Neither the local nor the global transition relations manipulate constraints. The constraints are constructed, manipulated and interpreted by defining the `trans/3` relation used by the model checker as shown in Figure 5.

The goal `trans((L1,Pi1), A, (L2,Pi2))`, given the set of clock valuations at the source location `L1` in `Pi1`, gives a transition label `A` and the set of clock valuations *at entry* to the destination location `L2`. Both `Pi1` and `Pi2` are represented as DBMs. We assume that the DBM `Pi1` is in canonical form whenever `trans/3` is invoked. The precondition for the `delay` operation is that `Pi1` be in

```
trans((L1, Pi1), A, (L2, Pi2)):-
        global_trans(L1, A, L2, Guards, Resets),
        delay(Pi1, Pi3),                    % Let all clocks progress
        invariants(L1, Inv1),               % as long as the
        add_constraints(Pi3, Inv1, Pi4),    % state invariants hold.
        add_constraints(Pi4, Guards, Pi5),  % Enfirce the guards and
        canonical_form(Pi5, Pi6),           % reduce to Canonical Form
        apply_resets(Pi6, Resets, Pi2),     % before resetting clocks.
        invariants(L2, Inv2),               % Impose invariants on the
        check_invariants(Pi2, Inv2).        % destination state.
```

Fig. 5. Computing the transition relation using DBM operations

canonical form. Hence we obtain the canonical form requirement as a precondition for `trans/3`. The goal `invariants(L1,Inv1)` simply picks up the location invariants at `L1`. The invariants of the source location are enforced, using `add_constraints` before checking the guards on the transition.

Note that the DBM `Pi4` at this point may not be canonical, and the operation to enforce the guards does not need `Pi4` to be in canonical form. The resulting DBM `Pi5` obtained may not be in canonical form either. However, the next operation, to reset clocks specified on the transition, needs the DBM to be in canonical form. Therefore we explicitly reduce the DBM `Pi5` to canonical form. The operation `apply_resets` sets the output DBM, `Pi2` in canonical form, which simplifies the last operation `check_invariants`. Moreover, we obtain the following pre- and post-conditions for computing `trans`: the input DBM must in canonical form, and the output DBM *will* be in canonical form. We can show by induction on derivations that if the DBM in the initial model checking query is in canonical form, the preconditions for every application of `trans` will be met.

In this sequence of DBM manipulations, we avoid unnecessary canonical form reductions, as observed in [25]. In previous implementations of real-time model checkers using DBM data structure, certain operations (such as `apply_resets`) were defined so that they preserve canonical forms. Computation of canonical form is a very expensive operation and avoiding them wherever possible will accrue gains in performance. What is novel in our implementation is that we not only deploy the above techniques to avoid unnecessary canonical form reductions, but also tolerate DBMs that are not in canonical form whenever possible. (See *Avoiding canonical form computation* in Section 5.)

3.3 Optimization of Constraint Operations

Among the constraint operations described above, `canonical_form` is the most expensive. We have already seen how the sequence of constraint operations can be carefully crafted so as to minimize the number of times canonical forms are explicitly constructed. Complementing this reduction, we lower the overheads for canonical form computation by exploiting the memo tables: by simply tabling `canonical_form`. As the performance figures presented in Section 4 show, the

sharing of canonical form computations significantly reduces model checking time. (See *Sharing canonical form computation* in Section 5.)

We observed that the same DBMs occur in multiple calls to `models` but with different locations. Hence each occurrence is separately tabled. After tabling the canonical form reduction, DBMs may be stored in multiple tables, and in call as well as answer tries. Since the terms representing DBMs are often large, and since DBMs contain no logical variables, we maintain a separate dictionary of DBMs, assigning each DBM a unique index, and use this index in all the tables. The time performance worsens by about 10% due to the indirection involved in accessing a DBM. However the space reductions are significant enough that it is worth the trade-off in time. (See *Dictionary representation* in Section 5.)

4 Experimental Results

In this section, we report on the comparative performance of XMC/dbm in two broad categories: (i) with respect to other real-time model checkers (XMC/RT, Uppaal and HyTech) to measure the effectiveness of XMC/dbm in practice; and (ii) with respect to XMC for model checking untimed systems to measure the overheads due to the constraint operations.

4.1 Real-Time Model Checking with XMC/dbm

We use as benchmarks two example systems, namely, Fischer's mutual exclusion protocol and the bridge-crossing system. (Details can be found in [9].) Table 1 presents the time (in seconds) taken by XMC/dbm, XMC/RT and HyTech for verifying several properties of the two benchmark programs. The measurements were done on a Sun Enterprise 4000 running Solaris 5.2.6 with 2GB of memory.

We verified safety, possibility and liveness properties. The first two are specified using timed modal mu-calculus and the much simpler reachability formula (denoted "mu-calc" and "reach" respectively, in the table) whereas liveness can only be specified in mu calculus. An entry "-" in the table means either that the property cannot be specified in that verification system (e.g., liveness properties with the reachability checker) or that the results could not be collected (e.g., possibility property for any of the systems for 6-processor Fischer's protocol). Observe that not only is XMC/dbm faster (by up to a factor of three) over XMC/RT, it also scales better. Moreover, XMC/dbm is competitive to HyTech.

We also measured the performance of Uppaal version 3.2.4 for verifying the safety property of Fischer's protocol on Intel Xeon 1.7GHz system with 1GB memory running Linux Mandrake 8.0. We measured the CPU time given by the operating system since the tool does not provide timing information. For 6 process Fischer, Uppaal takes 121 seconds whereas XMC/dbm takes 73 seconds.

In XMC/RT, the loose coupling of the POLINE constraint solver with XSB precluded it from any reasonable memory management. Hence we do not compare its memory usage with XMC/dbm. For 4,5 and 6 process Fisher XMC/dbm

Table 1. Comparative performance of XMC/dbm on real-time systems

System	Property	XMC/dbm		XMC/RT		HyTech
		mu-calc	reach	mu-calc	reach	
Fischer 3 proc	Safety	0.61	0.49	0.84	0.46	0.99
	Possibility	3.7	3.1	4.13	4.08	-
	Liveness	0.68	-	1.54	-	-
Fischer 4 proc	Safety	5.2	4.27	11.4	9.0	8.7
	Possibility	62.1	36.8	105	201	-
	Liveness	11.8	-	33.4	-	-
Fischer 5 proc	Safety	46	39.7	-	-	74.62
	Possibility	681.1	412.6	-	-	-
	Liveness	211.3	-	-	-	-
Fischer 6 proc	Safety	468.8	407.8	-	-	562.27
	Possibility	-	-	-	-	-
	Liveness	-	-	-	-	-
Bridge Crossing	Safety	0.07	0.06	0.19	0.17	0.14
	Bounded Liveness	2.1	0.4	7.0	1.19	1.19
	Liveness	0.2	-	0.8	-	-

requires 3.8, 28 and 386 MB respectively. HyTech needs 11, 93 and 678 MB respectively. Uppaal shows much better memory performance: e.g. for 6 processes it takes only 58 MB. Reducing the memory usage by using alternative encodings of DBMs is a topic of current study.

We measured the relative performance of XMC/dbm on the Philips audio control protocol, a real-life problem also widely reported in the literature [3]. The protocol specifies communication between devices over an interface bus without a central controller. We verified two prperties of the protocol in XMC/dbm. The safety property that the receiver never enters error state was verified by XMC/dbm in 18.4 seconds using 23 MB of memory. The other correctness property that the receiver receives entire message was verified in 35.3 seconds using 42 MB of memory. Both measurements were made on an Intel Xeon 1.7GHz system with 1GB memory running Linux Mandrake 8.0.

4.2 Verification of Untimed Systems

For measuring the overheads incurred by XMC/dbm when model checking untimed systems we use examples drawn from XMC's benchmark suite: `leader` which is a distributed leader election protocol, and `sieve` which implements a concurrent Sieve of Eratosthenes, were originally from the SPIN [14] example suite; and `metalock` is a protocol used to control access to object lock queues in Java, originally verified in XMC [2]. Table 2 shows the performance of XMC/dbm and XMC on these three examples. In the table the column "Time" shows running time in seconds and the column "Space" shows memory usage

Table 2. Comparative performance of XMC/dbm on untimed systems

System	Property	Size	Time		Space	
			XMC	XMC/dbm	XMC	XMC/dbm
Leader	one_leader	5	1.8	2.2	2.0	3.1
		6	10.2	12.4	5.6	8.8
		7	55.2	69.7	21.2	34.1
Sieve	ae_finish	(6,7,41)	0.9	1.1	1.7	2.6
		(6,8,43)	0.9	1.2	1.7	2.6
		(6,9,47)	1.0	1.3	1.8	2.7
Metalock	mutex	(1,4)	2.5	3.1	2.1	3.4
		(3,1)	1.6	2.2	2.5	3.9
		(2,2)	2.4	3.0	2.4	3.7

in MB. The performance measurements were done on an Intel Xeon 1.7MHz system with 1GB memory running Linux Mandrake 8.0.

In terms of time, XMC/dbm incurs an overhead of at most 30% for model checking untimed systems. The memory used by XMC/dbm is about 60% more than that of XMC. This is due to the `models/3` predicate in XMC/dbm where every state gets recorded twice for each formula: once as the first argument (in the query), and once as the third argument (in the answer). In contrast, using `models/2` in XMC, a state is saved only once for each formula, in the call table alone. In XMC, a query to `models` simply succeeds or fails, and hence nothing is stored in the answer tables.

5 Impact of Optimizations: Experimental Evaluation

Here we present the effect of optimizations described in Section 3:

Sharing Expensive Computations: Recall that computing canonical forms dominates the cost of model checking real-time systems. Therefore caching is critical for improving the performance. A unique aspect of XMC/dbm is to use XSB's tables for caching the results of canonical form computation. This has improved the performance of XMC/dbm by a factor of 2.5. For example, when caching of canonical form computation is verifying the reachability formula in for Fischer's protocol with 3–6 processors takes 0.49, 4.27, 39.7 and 407.8s resp., while it takes 0.76, 9.11, 103 and 1174 resp. when the optimization is disabled.

Avoiding Canonical Form Computation: Recall that a unique aspect of our implementation is that we can avoid canonical form computation immediately after applying the invariants at the source state. This optimization is effective whenever there are location invariants in the transition system. Fischer's protocol has only one location with an invariant, and the effect of this optimization is not measurable. In contrast, the Philips audio protocol specifies several location invariants, and the optimization improves verification times by a factor of nearly 2.5: 18.2s and 35.3s for the safety and correctness properties resp. with the optimization turned on, compared to 45.2s and 92.1s, resp., without optimization.

Dictionary of DBMs: Recall from Section 5 that storing DBMs in a dictionary structure can improve memory usage. With this optimization, we observe total memory usage of 1.6, 9.5 and 76 MB for verifying the safety property (by reachability) for Fischer's protocol with 4, 5 and 6 processes respectively. In contrast, without this optimization, the memory usage was 3.8, 28 and 386 MB respectively. Hence the dictionary structure reduces space requirement by more than a factor of three.

6 Conclusions

We have shown how a lightweight constraint solver combined with tabling and carefully optimized, can yield an efficient model checker for real-time systems. We showed, via experimental evaluation, that it is possible to implement such a model checker (XMC/dbm) without unduly degrading the performance of model checking untimed systems. Such a model checker forms a basis for creating an uniform environment to verify timed as well as untimed systems. The use of model checkers such as XMC/dbm will become important for verifying embedded systems which contain both real-time and non-real-time components. The application of XMC/dbm to such problems is the topic of future work.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 102
2. S. Basu, S. A. Smolka, and O. R. Ward. Model checking the Java Meta-Locking algorithm. In *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)*, Edinburgh, Scotland, April 2000. 111
3. J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using Uppaal. In *Proceedings of the 8th International Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 244–256, New Brunswick, New Jersey, USA, 1996. Springer-Verlag. 111
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1999. 108
5. G. Delzanno and S. Etalle. Transforming a proof system into prolog for verifying security protocols. In *International Workshop on Logic-based Program Synthesis and Transformation*, November 2001. 100
6. G. Delzanno, S. Mukhopadhyay, and A. Podelski. Constraint-based model checking for timed systems with accurate widenings. Technical Report, Max-Planck-Institut für Informatik, 1999. 100, 101
7. G. Delzanno and A. Podelski. Model checking in CLP. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), LNCS*, volume 1579, pages 223–239, Amsterdam, March 1999. 100, 101
8. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of CAV'89*. LNCS 407, 1989. 101, 106

9. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: a recipe for model checking real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 2000. 101, 104, 105, 106, 110
10. F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of sets of infinite state processes using program transofmration. In *International Workshop on Logic-based Program Synthesis and Transformation*, November 2001. 100
11. G. Gupta and E. Pontelli. A Constraint Based Approach for Specification and Verification of Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997. 100, 101
12. N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997. 101
13. T. A. Henzinger, P. H. Ho, and H. Wong-Toi. "HyTech: A model checker for hybrid systems". *International Journal on Software Tools for Technology Transfer*, 1(2):110–122, October 1997. 102
14. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. 111
15. K. G. Larsen, P. Pettersson, and W. Yi. Model checking for real-time systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in LNCS, pages 62–88, August 1995. 102
16. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997. 101
17. M. Leuschel and S. Gruner. Abstract partial deduction using regular types and its application to model checking. In *International Workshop on Logic-based Program Synthesis and Transformation*, November 2001. 100
18. U. Nilsson and J. Lubcke. Constraint logic programming for local and symbolic model-checking. In *In Proc. of the Int'l Conf. on Computational Logic (CL2000)*, volume 1861. Springer-Verlag, 2000. 100, 101
19. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 143–154, Haifa, Israel, July 1997. Springer-Verlag. 100
20. C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV '00)*, pages 576–580. Springer-Verlag, 2000. 101, 105
21. A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *Proceedings of the 13th International Conference on Computer-Aided Verification (CAV '01)*, volume 2102 of *LNCS*. Springer-Verlag, 2001. 100
22. O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings of the 7th International Conference on Computer-Aided Verification*, volume 939 of *LNCS*. American Mathematical Society, 1995. 104
23. L. Urbina. Analysis of hybrid systems in CLP(R). In *Constraint Programming (CP'96)*, volume LNCS 1102. Springer-Verlag, 1996. 100
24. XSB. The XSB logic programming system.
Available at `www.cs.sunysb.edu/~sbprolog`. 100
25. S. Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996. 108, 109

Constraint-Based Infinite Model Checking and Tabulation for Stratified CLP

Witold Charatonik[1,2], Supratik Mukhopadhyay[1,3⋆], and Andreas Podelski[1]

[1] Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
{witold,supratik,podelski}@mpi-sb.mpg.de
[2] University of Wrocław, Poland
[3] University of Pennsylvania

Abstract. Forward analysis procedures for infinite-state systems such as timed systems were limited to safety properties. We give the first constraint-based forward analysis for infinite-state systems that goes beyond safety properties. Namely, we take the restriction of the μ-calculus to least-fixpoint formulas where negation is applied to closed subformulas only. We characterize these properties as perfect models of constraint logic programs, and we present a tabulation procedure for the top-down evaluation of stratified constraint logic programs.

1 Introduction

Symbolic model checking for systems with variables over an infinite numeric domain, e.g. timed or hybrid systems, integer-valued network protocols or models of parametrized systems, is an important topic of research (see e.g. [1, 2, 3, 4, 14, 13, 9, 10, 16, 15, 18, 24, 25, 23, 27, 28, 38]). Here, 'symbolic' refers to the representation of a generally infinite set of states by a disjunctive constraint, i.e. a disjunction of conjunctions of atomic formulas (such as inequalities $x \leq y + c$ for timed systems).

One distinguishes between forward and backward methods. For example, for reachability properties ("no bad state can be reached"), the backward analysis starts with a constraint representing the 'bad' states and iteratively (1) goes through the disjunction (the 'database'), (2) for each constraint computes new constraints (representing predecessor states) and (3) adds the new constraints to the disjunction. The forward analysis is defined accordingly; it starts with initial states and iteratively computes successor states. The backward method extends directly to more general temporal logic properties defined e.g. in the μ-calculus. In practice (for example in the tool UppAal [7] for timed systems), the forward method is preferred because it does not explore unreachable states.

⋆ Support from the grants NSF award CCR99-70925, SRC award 99-TJ-688, and DARPA ITO Mobies award F33615-00-C-1707 is gratefully acknowledged.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 115–129, 2002.
© Springer-Verlag Berlin Heidelberg 2002

So far, forward methods for infinite-state systems have been developed for *safety* properties only. It seems difficult to find such methods for existing temporal logics such as Computation Tree Logic (CTL), the Alternation-Free μ-calculus AFμ, and more expressive fragments of μ-calculus. The difficulty arises with the operators AX and EG in CTL and the greatest-fixpoint operator in AFμ together with constraints.

We give a forward method for a new temporal logic that we call Sμ. It does not contain any of the above-mentioned operators, but it subsumes safety properties and it can express interesting liveness properties such as *convergence* (a.k.a. zenoness or timelock) [24, 38] for timed automata.

Sμ is the subset of all μ-calculus formulas (built up with the Boolean operators, the existential predecessor operator EX and and the least fixpoint combinator μ) where negation is applied to closed subformulas only.

We present the forward method for Sμ as a tabulation procedure for the top-down evaluation of stratified constraint logic programs. This is possible because we characterize Sμ properties as perfect models of such programs.

In the next section, we define Sμ as a subset of AFμ and investigate its expressiveness. In fact, we have a hierarchy where two levels correspond to already existing subsets of branching time logic, namely STL (see e.g. [5] and EF-Logic (see e.g. [18]), and the third, most general one corresponds to Sμ. We also show that, for finite systems, the lower bounds of the model checking problem for Sμ and AFμ are the same. In Section 3 we express Sμ properties in terms of the perfect model semantics for stratified constraint logic programs. In Section 4 we present a tabled-resolution procedure for computing this semantics and prove its correctness. Finally, in Section 5 we discuss the relation of our work with existing work and the *convergence* problem for timed automata.

2 The Temporal Logic Sμ

We first give the syntax of (closed) formulas φ of the modal μ-calculus and then define its subset Sμ.

$$\varphi \ ::= \ p \mid x \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \Diamond\varphi \mid \mu x.\, \varphi$$

Here p is an atomic proposition, x is a variable, $\Diamond$ is the next operator (written EX in CTL syntax), and in $\mu x.\varphi$, φ is monotone in x (i.e., all occurrences of the free variable x in φ lie in the scope of an even number of negations), and the least fixed point operator μx is considered to be a binder for the variable x. A formula φ is closed if it does not contain free variables. If we require that x does not occur free in any negative occurrence of a subformula $\neg\mu y.\psi$ of $\mu x.\, \varphi$, we obtain the *alternation-free* fragment of the μ-calculus AFμ.

Sometimes we use the formulas $\nu x.\varphi(x)$ and $\Box\varphi$ informally as abbreviations for $\neg\mu x.\neg\varphi(\neg x)$ and $\neg\Diamond\neg\varphi$, respectively.

Definition 1 (Sμ). *An Sμ formula is a closed formula φ of the modal μ-calculus such that in any subformula $\neg\psi$ of φ, the formula ψ is closed.*

The above definition immediately implies that the logic Sμ is a fragment of alternation-free μ-calculus. The example of the formula $\mu x.p \vee \neg\Diamond\neg x$ (abbreviated $\mu x.p \vee \Box x$, and written $\mathsf{AF}(p)$ in CTL syntax) of the alternation-free μ-calculus shows that Sμ is strictly less expressive than AFμ.

The Fisher-Ladner closure $cl(\varphi)$ of a formula φ is the smallest set of formulas containing φ such that

- if $\psi \in cl(\varphi)$ and ψ' is a subformula of ψ then $\psi' \in cl(\varphi)$, and
- if $\mu x.\psi(x) \in cl(\varphi)$ then $\psi(\mu x.\psi(x)) \in cl(\varphi)$.

Here, for a formula $\mu x.\psi(x)$, the formula $\psi(\mu x.\psi(x))$ is obtained from $\psi(x)$ by replacing each free occurrence of x with $\mu x.\psi(x)$.

Proposition 1. *Let φ be a modal μ-calculus formula. The following are equivalent.*

- *φ is an Sμ formula,*
- *the is-subformula relation over $cl(\varphi)$ can be made a partial order $\preceq$ that is strict wrt. to negation, i.e., $\varphi' \prec \neg\varphi'$,*
- *there exists a* stratification *function $\mathcal{S}$ assigning a natural number to each formula in the closure $cl(\varphi)$ such that*
  - *if $\varphi'' \in cl(\varphi')$ then $\mathcal{S}(\varphi'') \leq \mathcal{S}(\varphi')$, and*
  - *$\mathcal{S}(\varphi') < \mathcal{S}(\neg\varphi')$ for all $\varphi' \in cl(\varphi)$.*

Proof. By inspection. □

Thus, Sμ consists of all *stratifiable* formulas of the modal μ-calculus (in the syntax given above, i.e., with least fixpoints and Boolean set operators including the complement, but *without* greatest fixpoints).

Example 1. Consider once more the formula $\mu x.p \vee \neg\Diamond\neg x$. To see that it is not stratified, suppose that there exists an appropriate stratification function $\mathcal{S}$. Let $\psi(x)$ be the formula $p \vee \neg\Diamond\neg x$. Then $\mathcal{S}(\mu x.\psi(x)) \geq \mathcal{S}(\psi(\mu x.\psi(x))) \geq \mathcal{S}(\neg\Diamond\neg\mu x.\psi(x)) > \mathcal{S}(\Diamond\neg\mu x.\psi(x)) \geq \mathcal{S}(\neg\mu x.\psi(x)) > \mathcal{S}(\mu x.\psi(x))$, which is a contradiction.

Figure 1 situates the expressiveness of Sμ relative to the alternation-free μ-calculus, CTL and STL (i.e., STL is a subset of both CTL and Sμ). We recall the definition of the syntax of STL and of its sublogic, the EF-Logic.

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \mathsf{EX}(\varphi) \mid \mathsf{EF}(\varphi) \qquad \text{(EF-Logic)}$$

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \mathsf{EX}(\varphi) \mid \varphi\mathsf{EU}\varphi' \qquad \text{(STL)}$$

The inclusion EF $\subseteq$ STL $\subseteq$ Sμ is obvious if we use the usual encoding of CTL in AFμ. To see that Sμ is incomparable with CTL in terms of expressiveness, observe that (1) every satisfiable formula of Sμ has a model whose paths are all finite; the CTL formula $\mathsf{EG}(p)$, for example, does not have such a model; (2) the Sμ formula $\mu x.p \vee \Diamond\Diamond x$ expresses the property "p is reachable in an even number of steps", which cannot be expressed in CTL.

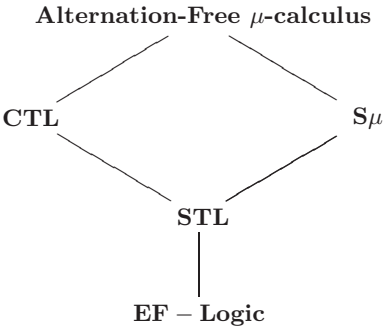


Fig. 1. A lattice for situating the expressiveness of Sμ and STL

Our motivation for Sμ stems from infinite-state systems with generally undecidable model checking problems. In fact, the following proposition shows that the restriction of the alternation-free μ-calculus to Sμ does not trade with a decrease of the theoretical complexity of model checking for finite-state systems. As in the alternation-free μ-calculus, the problem is hard for P even if the formula is fixed. The existing P-hardness proofs for the alternation-free μ-calculus, however, reduce the alternating graph-reachability problem, using the $\Box$ modality in an essential way, and cannot be carried over directly. Our proof is close in spirit to the proof of the P-completeness of Datalog LIT in [22]; we also reduce the monotone circuit value problem (MCVP), a well-known P-complete problem.

Proposition 2 (Finite-state systems). *The program complexity of Sμ model checking is P-complete.*

Proof. An instance of MCVP is a sequence of variables $X_1, \ldots, X_n$ and boolean equations of the form $X_i = \mathit{true}$, $X_i = \mathit{false}$, $X_i = X_j \wedge X_k$ or $X_i = X_j \vee X_k$ where for all equation of the last two forms we have $i > \max(j, k)$, such that the value of a variable does not depend on itself. The question we ask is whether the value of X_n is *true*.

Given such an instance I of MCVP, we construct a Kripke structure $K = \langle S, \rightarrow, L\rangle$ as follows. We define the set of atomic propositions as $\{p, p_1, p_2\}$. The set S of states consists of the variables $X_1, \ldots, X_n$ and their copies (two for each variable) $X_1^1, X_1^2, \ldots, X_n^1, X_n^2$. The labeling function $L : S \rightarrow 2^{\{p,p_1,p_2\}}$ is defined as follows. $L(X_i) = \{p\}$ iff $X_i = \mathit{true}$ is in I. For all $i = 1, \ldots, n$ we define $L(X_i^1) = \{p_1\}$ and $L(X_i^2) = \{p_2\}$. The transition relation $\rightarrow$ is defined as follows. For all equations $X_i = X_j \vee X_k$ we have $X_i \rightarrow X_j$ and $X_i \rightarrow X_k$, and for all equations $X_i = X_j \wedge X_k$ we have $X_i \rightarrow X_i^1$, $X_i \rightarrow X_i^2$ $X_i^1 \rightarrow X_j$ and $X_i^2 \rightarrow X_k$. Now it is easy to see that the state X_n in the structure K satisfies the formula

$$\mu x.p \vee \Diamond x \vee (\Diamond(p_1 \wedge \Diamond x) \wedge (\Diamond(p_2 \wedge \Diamond x)))$$

if and only if the value of the variable X_n in the circuit is *true*. $\Box$

3 Translation

If we restrict the translation of [22] to Sμ, we can formulate it in terms of Stratified CLP (stratified constraint logic programs, see [26]). The parametric constraint system of the CLP paradigm is used to express the transition relation of the system to be checked (e.g. arithmetic constraints over reals for timed or hybrid systems).

We assume the definition of predicates $\mathsf{trans}(\mathbf{s}, \mathbf{s}')$, $\mathsf{init}(\mathbf{s})$ and $p_{ap}(\mathbf{s})$ expressing, respectively, that there is a transition from the state $\mathbf{s}$ to $\mathbf{s}'$, that $\mathbf{s}$ is an initial state, and that the atomic proposition ap holds in the state $\mathbf{s}$.

We translate an Sμ formula ψ to the constraint logic program $\mathcal{P}_\psi$ that contains all clauses defining the predicate trans and the clauses below defining new predicates p_φ, for each formula φ in the Fisher-Ladner closure $cl(\psi)$.

$$
\begin{array}{ll}
(1) & p_{\varphi_1 \wedge \varphi_2}(\mathbf{s}) \leftarrow p_{\varphi_1}(\mathbf{s}), p_{\varphi_2}(\mathbf{s}) \\
(2) & p_{\varphi_1 \vee \varphi_2}(\mathbf{s}) \leftarrow p_{\varphi_1}(\mathbf{s}) \\
 & p_{\varphi_1 \vee \varphi_2}(\mathbf{s}) \leftarrow p_{\varphi_2}(\mathbf{s}) \\
(3) & p_{\neg\varphi}(\mathbf{s}) \leftarrow \mathsf{not}(p_\varphi(\mathbf{s})) \\
(4) & p_{\Diamond\varphi}(\mathbf{s}) \leftarrow \mathsf{trans}(\mathbf{s}, \mathbf{s}'), p_\varphi(\mathbf{s}') \\
(5) & p_{\mu x.\varphi(x)}(\mathbf{s}) \leftarrow p_{\varphi(\mu x.\varphi(x))}(\mathbf{s})
\end{array}
$$

We can abbreviate the translation of formulas of EF-Logic or STL(i.e. without first expressing them as Sμ formulas).

$$
\begin{array}{ll}
(6) & p_{\mathsf{EF}\varphi}(\mathbf{s}) \leftarrow p_\varphi(\mathbf{s}) \\
 & p_{\mathsf{EF}\varphi}(\mathbf{s}) \leftarrow \mathsf{trans}(\mathbf{s}, \mathbf{s}'), p_{\mathsf{EF}\varphi}(\mathbf{s}') \\
(7) & p_{\varphi_1 \mathsf{EU} \varphi_2}(\mathbf{s}) \leftarrow p_{\varphi_2}(\mathbf{s}) \\
 & p_{\varphi_1 \mathsf{EU} \varphi_2}(\mathbf{s}) \leftarrow p_{\varphi_1}(\mathbf{s}), \mathsf{trans}(\mathbf{s}, \mathbf{s}'), p_{\varphi_1 \mathsf{EU} \varphi_2}(\mathbf{s}')
\end{array}
$$

The resulting constraint logic program is *stratified*, which means that there exists a mapping assigning each predicate p its *level* such that in every clause of the form $p(\boldsymbol{x}) \longleftarrow B \wedge \varphi$, the level of the predicate of an atom occurring positively in B is less than or equal to that of p, and the level of the predicate of an atom occurring negatively in B is strictly less than the level of p.

The original definition of a *perfect model* of a stratified constraint logic program $\mathcal{P}$ is model-theoretic [31]. In the proof of our theorem below, we use the following equivalent characterization of [6].

The perfect model $pm(\mathcal{P})$ of the stratified constraint logic program $\mathcal{P}$ with maximum predicate level k, is the set M_k in a sequence $M_{-1}, M_0, \ldots, M_k$ where
— $M_{-1} = \emptyset$, and
— M_j is the least fixpoint of $T^j_{\mathcal{P}}$, the restriction of the logical consequence operator $T_{\mathcal{P}}$ to the lattice C_j that contains M_{j-1} and all subsets of atoms with predicates of level j,

$$C_j = \{M_{j-1}\} \cup 2^{\{p(\boldsymbol{v}) \in B_{\mathcal{D}} \mid level(p) = j\}}.$$

Here, given the constraint domain $\mathcal{D}$ for the program $\mathcal{P}$, $B_{\mathcal{D}}$ denotes the $\mathcal{D}$-base $\mathcal{P}$ (in generalization of the Herbrand base). In the sequel, we assume that the constraint domain $\mathcal{D}$ admits elimination of existential and universal quantifiers.

We define the denotation of a predicate p in the perfect model of a program $\mathcal{P}$ by

$$[\![p]\!]_{pm(\mathcal{P})} = \{\boldsymbol{v} \mid p(\boldsymbol{v}) \in pm(\mathcal{P})\}.$$

Similarly, for other sets M, the denotation of p in M is $[\![p]\!]_M = \{\boldsymbol{v} \mid p(\boldsymbol{v}) \in M\}$. For negative literals $\neg p$ we define $[\![\neg p]\!]_{pm(\mathcal{P})} = \mathcal{D}^n - [\![p]\!]_{pm(\mathcal{P})}$ where n is the arity of the predicate p. We use a similar notation for the denotation of Sμ formulas: if a transition system is defined by a program $\mathcal{P}$ as above, the set of states satisfying the formula φ is denoted by $[\![\varphi]\!]_{\mathcal{P}}$.

Theorem 1 (Adequacy of Translation). *The translation maps a Sμ formula φ to a stratified constraint logic program $\mathcal{P}_\varphi$ such that the atom $p_\varphi(\mathbf{s})$ belongs to the perfect model of the program $\mathcal{P}_\varphi$ if and only if φ is true of the state $\mathbf{s}$; or: the denotation of the predicate p_φ in the perfect model semantics of the program $\mathcal{P}_\varphi$ is exactly the denotation of the Sμ formula φ wrt. the transition system defined by $\mathcal{P}$,*

$$[\![p_\varphi]\!]_{pm(\mathcal{P}_\varphi)} = [\![\varphi]\!]_{\mathcal{P}}.$$

4 Tabulation

We now present a tabulation procedure for the top-down query evaluation of a query for a stratified constraint logic program, which, by Theorem 1, is a local symbolic model checking procedure for Sμ formulas. The input is a query consisting of an atom $p(\boldsymbol{x})$ and a constraint Φ. Upon termination the answer is a *complete* list of answer constraints $[\Phi_1, \ldots, \Phi_n]$. This means: for every tuple $\boldsymbol{v}$ of values for the argument tuple $\boldsymbol{x}$, the atom $p(\boldsymbol{v})$ lies in the perfect model of $\mathcal{P}$ and $\boldsymbol{v}$ is a solution of the constraint Φ if and only if $\boldsymbol{v}$ is a solution of one of the answer constraints Φ_i. In short: the denotation of the predicate p in the perfect model semantics of the program $\mathcal{P}$ intersected with the set of solutions of Φ is the set of solutions of all answer constraints, i.e., of the disjunction $\Phi_1 \vee \ldots \vee \Phi_n$,

$$[\![p]\!]_{pm(\mathcal{P})} \cap [\![\Phi]\!] = [\![\Phi_1 \vee \ldots \vee \Phi_n]\!]$$

where for a constraint Φ its denotation is its set of solutions $[\![\Phi]\!] = \{\boldsymbol{v} \mid \boldsymbol{v} \models \Phi\}$.

If the program $\mathcal{P}$ arises from the translation given in the previous section (i.e., it is of the form $\mathcal{P}_\psi$ for an Sμ formula ψ) and the constraint Φ describes the set of initial states, then tabled resolution starting with the query $\langle p(\boldsymbol{x}), \Phi\rangle$ corresponds to forward analysis (depth-first or breadth-first, depending on the selection strategy of the resolution procedure; here, we consider a depth-first procedure). The answer constraints then specify which of the initial states satisfy the Sμ formula ψ (possibly all of them, namely if the disjunction is equivalent to Φ).

The procedure is based on *tabled resolution*. Originally tabled resolution was an execution strategy for definite logic programs [36]; it was extended to logic programs with negation in [33, 11], but for soundness reasons its use in the case of negative literals is limited to ground queries. In this paper we extend it to handle constraints and nonground constructive negation (i.e., constructive negation with nonground queries).

The central data structure is a table T of answer constraints. An index in the extendable table is a query of the form $\langle p'(\boldsymbol{x}), \Phi' \rangle$ (consisting of an atom $p'(\boldsymbol{x})$ and a constraint Φ'); the corresponding field contains a list of answers to that query that have been computed to far (we will often identify a list of constraints with the disjunction of these constraints). Initially, the table has only one index, namely the original query $\langle p(\boldsymbol{x}), \Phi \rangle$, whose field contains the empty list.

The procedure is defined as the iterative application of the rules in Figure 2 until a fixpoint is reached. We start with the initial query $\langle p(\boldsymbol{x}), \Phi \rangle$ by creating an index with this query in the table T. Iteratively, we apply resolution steps, hereby creating new indexes for the new queries and thus extending the table (for the two extension rules), and we store all answers to a query encountered so far in the table T (for the two completion rules). The stored answers are reused in further resolution steps. This goes on until a fixpoint is reached.

To force the termination of the computation on all programs where the usual resolution terminates as well as many other programs, we obey the following strategy.

- All constraints occurring in the table (both in the indexes and entries) must be satisfiable (unsatisfiable constraints are simply omitted),
- Redundant indexes are not created. If the table contains an index $(p(\boldsymbol{x}), \Psi)$ and $\Phi \models \Psi$ then the index $(p(\boldsymbol{x}), \Phi)$ is not created; that is, the respective extension rule is not applied.
- Redundant answers are not stored. If for some query Q the entry $T(Q)$ contains a constraint Ψ and $\Phi \models \Psi$ then the constraint Φ is not added to $T(Q)$; that is, the positive completion rule is not applied.
- For a given query $\langle \neg p(\boldsymbol{x}), \Phi \rangle$, the negative completion rule is applied only once.

We now discuss briefly the rules in Figure 2. The two positive rules extend (in higher-level presentation) the original tabling procedure [36] to handle constraints. Their application correspond directly to one step of resolution. In both cases the query $(p(\boldsymbol{x}), \Phi)$ is resolved with, respectively, the clauses $p(\boldsymbol{x}) \leftarrow \bigwedge_i l_i(\boldsymbol{y}_i) \wedge l(\boldsymbol{y}) \wedge R \wedge \Phi'$ and $p(\boldsymbol{x}) \leftarrow \bigwedge_i l_i(\boldsymbol{y}_i) \wedge \Phi'$. The answers to the calls of $l_i(\boldsymbol{y}_i)$ are not computed, but taken from the table. (Note that the conjunction $\bigwedge_i l_i(\boldsymbol{y}_i)$ may be empty; this accommodates the case where $l(\boldsymbol{y})$ is the first literal of the clause.) Here, R represents the rest (the conjunction of the remaining literals) of the body in the first clause. In the first case the resolution results in a new query, in the second case the result is an answer that is stored in the table (unless it is redundant).

The two negative rules are to handle constructive negation. In order to compute an answer to a negative query $\langle \neg p(\boldsymbol{x}), \Phi \rangle$ we first run the procedure for

(positive extension)

$$\frac{\begin{array}{l} \mathit{Index}(p(\boldsymbol{x}), \Phi) \\ p(\boldsymbol{x}) \leftarrow \bigwedge_i l_i(\boldsymbol{y}_i) \wedge l(\boldsymbol{y}) \wedge R \wedge \Phi' \in \mathcal{P} \\ \Phi_i \in T(l_i(\boldsymbol{y}_i), \Psi_i) \text{ where } \Phi \wedge \Phi' \models \Psi_i \text{ for all } i \end{array}}{\mathit{Index}(l(\boldsymbol{y}), \bigwedge_i \Phi_i \wedge \Phi \wedge \Phi')}$$

(positive completion)

$$\frac{\begin{array}{l} \mathit{Index}(p(\boldsymbol{x}), \Phi) \\ p(\boldsymbol{x}) \leftarrow \bigwedge_i l_i(\boldsymbol{y}_i) \wedge \Phi' \in \mathcal{P} \\ \Phi_i \in T(l_i(\boldsymbol{y}_i), \Psi_i) \text{ where } \Phi \wedge \Phi' \models \Psi_i \text{ for all } i \end{array}}{\exists_{-\boldsymbol{x}} \bigwedge_i \Phi_i \wedge \Phi \wedge \Phi' \in T(p(\boldsymbol{x}), \Phi)}$$

(negative extension)

$$\frac{\mathit{Index}(\neg p(\boldsymbol{x}), \Phi)}{\mathit{Index}(p(\boldsymbol{x}), \Phi)}$$

(negative completion)

$$\frac{\begin{array}{l} \mathit{Index}(\neg p(\boldsymbol{x}), \Phi) \\ \Phi \models \Psi \\ \mathit{Index}(p(\boldsymbol{x}), \Psi) \end{array}}{T(\neg p(\boldsymbol{x}), \Phi) = DNF(\Phi \wedge \neg T(p(\boldsymbol{x}), \Psi))} \text{ (provided } T(p(\boldsymbol{x}), \Psi) \text{ is fully computed)}$$

Fig. 2. Rules for the tabulation procedure

the positive query (it is important here that we collect *all* the answers for the positive query) and then negate the disjunction of the answers. Note that every entry in the table is a disjunction of constraints, so we have to bring the negation of the disjunction into disjunctive normal form. Note also that if the program is stratified, the computation for $p(\boldsymbol{x})$ never involves computing $\neg p(\boldsymbol{y})$ and thus the side condition "$T(p(\boldsymbol{x}), \Psi)$ is fully computed" is well-founded.

If the procedure terminates (which in general cannot be guaranteed for decidability reasons), then the table contains all answers to all sub-queries and to the original query $p(\boldsymbol{x})$.

The examples below give an intuition about the procedure. Example 2 shows how tabling helps avoiding infinite loops in resolution. Example 3 shows how answers to *all* subgoals are stored in the table (for all successful subderivations). Finally, Example 4 shows how the negation is handled.

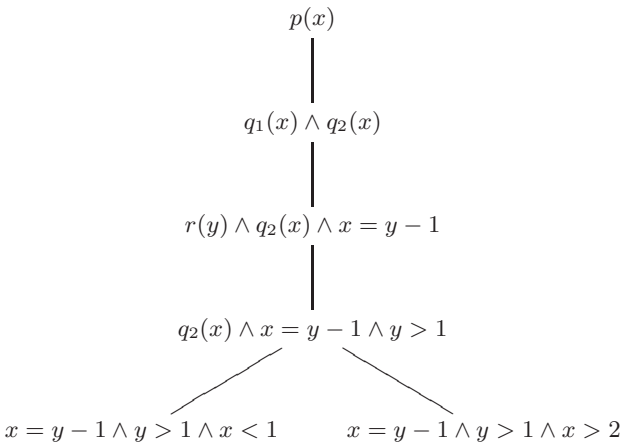


Fig. 3. Computation tree for Example 3

Example 2. Consider the query $p(x)$ for the following program.

$$\begin{aligned} p(x) &\leftarrow x = 0. \\ p(x) &\leftarrow x = 1. \\ p(x) &\leftarrow p(x). \end{aligned}$$

We start with the table consisting of one entry $T[p(x), \mathit{true}] = [\,]$. After a double application of the positive completion rule we obtain two answers $x = 0$ and $x = 1$. Now, an application of the positive extension rule to the clause $p(x) \leftarrow p(x)$ leads to a redundant index in the table; an application of the positive completion rule to this clause gives only redundant answers. Hence, at this point no more resolution steps are possible and the procedure terminates.

Example 3. Consider the query $p(x)$ for the following program.

$$\begin{aligned} p(x) &\leftarrow q_1(x), q_2(x). \\ q_1(x) &\leftarrow x = y - 1, r(y). \\ r(x) &\leftarrow x > 1. \\ q_2(x) &\leftarrow x < 1. \\ q_2(x) &\leftarrow x > 2. \end{aligned}$$

The derivation tree for this query is shown on Figure 3. In the fourth node of this tree we obtain an answer $x = y-1 \wedge y > 1$ to the subquery $r(y)$. This answer is normalized (the formula $\exists x.\ x = y - 1 \wedge y > 1$ is simplified by eliminating the existentially quantified x to the formula $y > 1$) and stored in the table: $T[\langle r(y), x = y - 1\rangle] = [y > 1]$.

Note that the constraint $x = y - 1 \wedge y > 1$ is not only an answer to the subquery $r(y)$, but also to the subquery $q_1(x)$. This is covered by the application of the positive completion rule to the query $\langle q_1(x), true\rangle$ and the clause $q_1(x) \leftarrow x = y - 1, r(y)$, using the already stored answer for $\langle r(y), x = y - 1\rangle$. The answer is again normalized (this time the variable y, and not x, is existentially quantified) and stored: $T[\langle q_1(x), true\rangle] = [x > 0]$.

The same happens when we reach leaves of the tree. The two answers that we obtain have to be stored both in the entry for $q_2(x)$ and $p(x)$, thus we obtain $[x > 0 \wedge x < 1, \quad x > 2]$ as the value for $T[\langle p(x), true\rangle]$ and $T[\langle q_2(x), x = y - 1 \wedge y > 1\rangle]$.

Example 4. Consider the program from Example 3 together with a clause $p'(x) \leftarrow \neg p(x)$ and a query $p'(x)$. Since the tree from Figure 2 cannot be extended anymore (it is totally successful in the terminology of Stuckey [35]), we can negate the answers collected for the query $p(x)$ and obtain correct answers for the query $\neg p(x)$ that (after transforming to the disjunctive normal form) we store in the table: $T[\langle \neg p(x), true\rangle] = [x \leq 0, \quad x \geq 1 \wedge x \leq 2]$. (Note: since we always eliminate existential quantifiers, negating constraints does not introduce universal quantification; i.e., we do not have to treat queries with universally quantified variables.)

We next investigate the correctness of the tabulation procedure.

Theorem 2 (Soundness and Completeness). *If the tabling procedure terminates then, for every query $\langle l(\boldsymbol{x}), \Phi\rangle$ that occurs as an index in the table T, the entry of T at that index is a list of constraints $[\Phi_1, \ldots, \Phi_n]$ such that*

$$[\![l]\!]_{pm(\mathcal{P})} \cap [\![\Phi]\!] = [\![\Phi_1 \vee \ldots \vee \Phi_n]\!].$$

That is, the denotation of the literal p in the perfect model semantics of the program $\mathcal{P}$ intersected with the set of solutions of Φ is the set of solutions of all constraints $\Phi_1, \ldots, \Phi_n$. Equivalently, for every solution $\boldsymbol{v}$ of the constraint Φ then the atom $p(\boldsymbol{v})$ lies (respectively, in the negated case, does not lie) in the perfect model of $\mathcal{P}$ and if and only if $\boldsymbol{v}$ is a solution of one of the constraints Φ_i.

Proof. The proof goes by the induction on the level of the literal $l(\boldsymbol{x})$. For this, we extend the definition of the level mapping from positive atoms to negative literals by putting $level(\neg p) = 1 + level(p)$.

First, recall that $pm(\mathcal{P}) = \bigcup_m M_m$ where $M_{m+1} = T_{\mathcal{P}}^{m+1} \uparrow \omega$, and that for predicates p of level m we have $[\![p]\!]_{pm(\mathcal{P})} \subseteq M_m$.

Suppose that the procedure terminates and that for all literals $l(\boldsymbol{x})$ of level less than or equal to m, where $m \geq -1$, and all constraints Φ such that $\langle l(x), \Phi\rangle$ is an index in the table we have

$$[\![l]\!]_{pm(\mathcal{P})} \cap [\![\Phi]\!] = [\![T(l(\boldsymbol{x}), \Phi)]\!].$$

Below we prove that this equality holds for all literals of level $m + 1$. Note that the case $m = -1$ covers the induction basis.

Take a table entry $T(l(\boldsymbol{x}), \Phi) = \bigvee_i \Phi_i$ where $l(\boldsymbol{x})$ has level $m+1$. Now we prove the theorem separately for the two cases, where $l(x)$ is positive and negative, respectively.

Case 1. $l(\boldsymbol{x})$ is a negative literal $\neg p(\boldsymbol{x})$.

For the $\subseteq$ direction (completeness of constructive negation), suppose $\boldsymbol{v} \in [\![\neg p]\!]_{pm(\mathcal{P})} \cap [\![\Phi]\!]$. Since $\langle \neg p(\boldsymbol{x}), \Phi\rangle$ is an index in the table and the procedure has terminated, we know that by the negative extension rule the table contains an index $\langle p(\boldsymbol{x}), \Psi\rangle$ such that $\Phi \models \Psi$. By the induction hypothesis $[\![p]\!]_{pm(\mathcal{P})} \cap [\![\Psi]\!] = [\![T(p(\boldsymbol{x}), \Psi)]\!]$. Now, $\boldsymbol{v} \in [\![\neg p]\!]_{pm(\mathcal{P})} = \mathcal{D}^n - [\![p]\!]_{pm(\mathcal{P})}$ where n is the arity of p. Therefore, $\boldsymbol{v} \not\in [\![p]\!]_{pm(\mathcal{P})}$ and so $\boldsymbol{v} \not\in T(p(\boldsymbol{x}), \Psi)$. Thus, $\boldsymbol{v} \in [\![\Phi \wedge \neg T(p(\boldsymbol{x}), \Psi)]\!] = [\![T(\neg p(\boldsymbol{x}), \Phi)]\!]$.

For the $\supseteq$ direction (soundness of constructive negation), suppose $\boldsymbol{v} \in [\![T(\neg p(\boldsymbol{x}), \Phi)]\!]$. Then the entry $T(\neg p(\boldsymbol{x}), \Phi)$ is nonempty, and by the termination assumption the table must contain an index $\langle p(\boldsymbol{x}), \Psi\rangle$ such that $\Phi \models \Psi$ and $[\![T(\neg p(\boldsymbol{x}), \Phi)]\!] = [\![\Phi \wedge \neg T(p(\boldsymbol{x}, \Psi)]\!]$. Then $\boldsymbol{v} \in \Phi$ and $\boldsymbol{v} \not\in [\![T(p(\boldsymbol{x}, \Psi)]\!]$. By the induction hypothesis, $[\![T(p(\boldsymbol{x}), \Psi)]\!] = [\![p]\!]_{pm(\mathcal{P})} \cap [\![\Psi]\!]$. Since $\Phi \models \Psi$, we have $[\![\Phi]\!] \subseteq [\![\Psi]\!]$ and $[\![p]\!]_{pm(\mathcal{P})} \cap [\![\Phi]\!] \subseteq [\![p]\!]_{pm(\mathcal{P})} \cap [\![\Psi]\!]$. Therefore $\boldsymbol{v} \not\in [\![p]\!]_{pm(\mathcal{P})} \cap [\![\Phi]\!]$. But $\boldsymbol{v} \in [\![\Phi]\!]$, so $\boldsymbol{v} \not\in [\![p]\!]_{pm(\mathcal{P})}$. Thus $\boldsymbol{v} \in [\![\neg p]\!]_{pm(\mathcal{P})} \cap [\![\Phi]\!]$.

Case 2. $l(\boldsymbol{x})$ is positive literal $p(\boldsymbol{x})$.

We start with the $\subseteq$ direction (completeness for the positive part). Here the proof goes by the induction on the number of iterations of the $T_{\mathcal{P}}^{m+1}$ operator. We show by an induction on n that for all predicates p of level $m+1$ and all constraints Φ such that $\langle p(\boldsymbol{x}), \Phi\rangle$ is an index in the table T

$$[\![p]\!]_{(T_{\mathcal{P}}^{m+1})^n(M_m)} \cap [\![\Phi]\!] \subseteq [\![T(p(\boldsymbol{x}), \Phi)]\!]$$

For $n = 0$ (the induction basis) this is trivially satisfied since $(T_{\mathcal{P}}^{m+1})^0(M_m) = M_m$ and M_m does not contain any atoms of level greater than m.

Now consider the case $n > 0$. Let $\boldsymbol{v} \in [\![p]\!]_{(T_{\mathcal{P}}^{m+1})^n(M_m)} \cap [\![\Phi]\!]$. By the definition of the $T_{\mathcal{P}}^{m+1}$ operator, there exists a clause $C \in \mathcal{P}$ of the form $p(\boldsymbol{x}) \leftarrow \bigwedge_i l_i(\boldsymbol{y}_i) \wedge \Phi'$ and tuples of elements $\boldsymbol{v}_i$ from the domain $\mathcal{D}$ such that $\boldsymbol{v}, \{\boldsymbol{v}_i\}_i \models \Phi \wedge \Phi'$ and $(T_{\mathcal{P}}^{m+1})^{n-1}(M_m) \models \bigwedge_i l_i(\boldsymbol{v}_i)$. By several applications of the positive extension rules (each for the clause C and some of $l_i(\boldsymbol{y}_i)$) we have that there are indexes $\langle l_i(\boldsymbol{y}_i), \Psi_i\rangle$ in the table T such that $\boldsymbol{v}_i \models \Psi_i$. By the induction hypothesis $\boldsymbol{v}_i \in [\![T(l_i(\boldsymbol{y}_i), \Psi_i)]\!]$. Hence, there exist constraints $\Phi_i \in T(l_i(\boldsymbol{y}_i), \Psi_i)$ such that $v_i \models \Phi_i$. Then $\boldsymbol{v}, \{\boldsymbol{v}_i\}_i \models \bigwedge_i \Phi_i \wedge \Phi \wedge \Phi'$, and thus $\boldsymbol{v} \models \exists_{-\boldsymbol{x}} \bigwedge_i \Phi_i \wedge \Phi \wedge \Phi'$. Now an application of the positive completion rule shows that $\boldsymbol{v} \in [\![T(p(\boldsymbol{x}), \Phi)]\!]$.

The $\supseteq$ direction (soundness of the positive part) is proved in a quite similar way, by induction on the number of applications of the positive completion rule to the atoms of level $m+1$.

We prove that for all $n \geq 0$

$$[\![p]\!]_{pm(\mathcal{P}))} \cap [\![\Phi]\!] \supseteq [\![T_{(n)}(p(\boldsymbol{x}), \Phi)]\!]$$

where $T_{(n)}$ is the table T after n applications of the positive completion rule to a predicate of level $m+1$.

For $n = 0$ this is trivially satisfied, since $T_{(0)}(p(\boldsymbol{x}), \Phi)$ is empty.

For $n > 0$, suppose $\Psi \in T_{(n)}(p(\boldsymbol{x}), \Phi)$ and $\boldsymbol{v} \in [\![\Psi]\!]$. Since Φ is a conjunct in Ψ, we have $\boldsymbol{v} \in [\![\Phi]\!]$, so we only have to prove that $\boldsymbol{v} \in [\![p]\!]_{pm(\mathcal{P}))}$. If $\Psi \in T_{(n-1)}(p(\boldsymbol{x}), \Phi)$ then we are done by the induction hypothesis, so suppose $\Psi \notin T_{(n-1)}(p(\boldsymbol{x}), \Phi)$. Then there exists a clause $p(\boldsymbol{x}) \leftarrow \bigwedge_i l_i(\boldsymbol{y}_i) \wedge \Phi'$ and constraints Φ_i, Ψ_i such that $\Phi_i \in T_{(n-1)}(l_i(\boldsymbol{y}_i), \Psi_i)$ and $\Psi = \exists_{-\boldsymbol{x}} \bigwedge_i \Phi_i \wedge \Phi \wedge \Phi'$. Since Ψ is satisfiable, we can find $\boldsymbol{v}_i$ such that $\boldsymbol{v}, \{\boldsymbol{v}_i\}_i \models \bigwedge_i \Phi_i \wedge \Phi \wedge \Phi'$. By the induction hypothesis (since all l_i are of level $\leq m + 1$), $M_{m+1} \models l_i(\boldsymbol{v}_i)$. Therefore $p(\boldsymbol{v}) \in T_{\mathcal{P}}^{m+1}(M_{m+1}) \subseteq pm(\mathcal{P})$. □

5 Related Work, Future Work

The logic-programming approach to verification is an active topic of research; see e.g. [10, 16, 20, 21, 22, 29, 30, 32]. The work that is directly related to ours is [32] and [29], where the authors use logic programming and tabulation to compute CTL properties of *finite*-state systems. One may characterize the difference between that work and ours by stating that we trade the loss of expressiveness of the considered temporal logic (Sμ vs. CTL) with the gain in generality of the systems to be verified (infinite vs. finite).

In [22], Gottlob, Grädel and Veith consider two linear-time, P-complete versions of Datalog, Datalog LIT and its Extension Datalog LITE, and translate formulas of the Alternation-Free μ-calculus AFμ into Datalog LITE in such a way that backward model checking amounts to bottom-up query evaluation. No top-down query evalution procedure is known for Datalog LITE over infinite models. Sμ is exactly the fragment of AFμ such that the translation of [22] falls into Datalog LIT; their translation then becomes the one presented in Section 3.

In a quite extensive literature on the subject, we found many tabulation procedures that are superficially related (see e.g. [8, 11, 12, 17, 19, 33, 34, 37, 36]); since we did not not find one that fits exactly, we felt that we had to present it here.

Two seemingly unrelated algorithms for detecting convergence of timed automata have been proposed. One is based on backward analysis [24], the other on forward analysis [38]. Once we have observed that convergence can be expressed in Sμ, in fact already EF-Logic (for every timed automaton $\mathcal{A}$, one can construct a timed automaton $\mathcal{A}'$ (in linear time) such that $\mathcal{A}$ is convergent if and only if the EF-Logic-formula $\mathsf{EF}\neg\mathsf{EF}(y > 1)$ is true for the automaton $\mathcal{A}'$), it is not difficult to see that the two algorithms are ad-hoc solutions for an instance of the problem of computing a perfect model of a stratified constraint logic program, and that they correspond to the bottom-up resp. top-down approach.

We leave it for future work to make precise our intuition that Sμ captures the boundary in expressiveness of temporal logic properties for which constraint-based forward methods exist. Also, it is open whether the equality STL=CTL$\cap$Sμ holds wrt. expressiveness (i.e. not only syntactically via the embedding of CTL and STL into AFμ).

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model checking in dense real time. *Information and Computation*, 104(2):2–34, 1993. 115

[2] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R. Grossman, A. Nerode, A. Ravn, and H. Rischel, editors, *Hybrid Systems I*, LNCS 736, pages 209–229. Springer-Verlag, 1993. 115

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, 1994. 115

[4] R. Alur and T. A. Henzinger. Modularity for timed and hybrid systems. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR'97: Concurrency Theory*, volume 1243 of *LNCS*, pages 74–88. Springer-Verlag, 1997. 115

[5] R. Alur and T. A. Henzinger. Computer-aided verification: An introduction to model building and model checking for concurrent systems, 1999. Book in preparation. 116

[6] K. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–214. Morgan Kaufmann, 1988. 119

[7] J. Bengtsson, K. G. Larsen, F. Larsson, P. Petersson, and W. Yi. Uppaal in 1995. In T. Margaria and B. Steffen, editors, *TACAS*, LNCS 1055, pages 431–434. Springer-Verlag, 1996. 115

[8] D. Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *the fifth International Symposium on Logic Programming*, pages 111–125. MIT Press, 1988. 126

[9] W. Charatonik, D. McAllester, D. Niwinski, A.Podelski, and I. Walukiewicz. The Horn mu-calculus. In V. Pratt, editor, *The 13th IEEE Annual Symposium on Logic in Computer Science*, 1998. 115

[10] W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 358–375, Lisbon, Portugal, March-April 1998. Springer-Verlag. 115, 126

[11] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1):20–74, 1996. 121, 126

[12] P. Codognet. A tabulation method for constraint logic programming. Proceedings of INAP'95, Industrial Applications of Prolog, Tokyo, Japan, 1995. 126

[13] H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis, and Presburger Arithmetics. In A. J. Hu and M. Y. Vardi, editors, *CAV'98: Computer Aided Verification*, volume 1427 of *LNCS*, pages 268–279. Springer-Verlag, 1998. 115

[14] B. Cui, Y. Dong, X. Du, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *PLAP/ALP98*, volume 1490 of *LNCS*, pages 1–20. Springer-Verlag, 1998. 115

[15] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In B. Steffen, editor, *TACAS98: Tools and Algorithms for the Construction of Systems*, LNCS 1384, pages 313–329. Springer-Verlag, March/April 1998. 115

[16] G. Delzanno and A. Podelski. Model Checking in CLP. In R. Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for*

Construction and Analysis of Systems (TACAS'99), volume 1579 of *LNCS*, pages 223–239, Amsterdam, The Netherlands, January 1999. Springer-Verlag. 115, 126

[17] X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *Proceedings of RTSS 2000*, 2000. 126

[18] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997. 115, 116

[19] F. Fages. Constructive negation by prunning. *Journal of Logic Programming*, 32(8):85–118, August 1997. 126

[20] L. Fribourg and M. V. Peixoto. Concurrent constraint automata. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, page 656, Vancouver, Canada, 1993. The MIT Press. 126

[21] L. Fribourg and J. Richardson. Symbolic verification with gap-order constraints. In J. P. Gallagher, editor, *LOPSTR'96: Logic Based Program Synthesis and Transformation*, volume 1207 of *LNCS*, pages 20–37. Springer-Verlag, 1996. 126

[22] G. Gottlob, E. Grädel, and H. Veith. Linear Time Datalog for Branching Time Logic. In J. Minker, editor, *Logic-Based Artificial Intelligence*, chapter 19. Kluwer, 2000. 118, 119, 126

[23] G. Gupta and E. Pontelli. A constraint-based approach for the specification and verification of real-time systems. In K.-J. Lin, editor, *IEEE Real-Time Systems Symposium*, pages 230–239. IEEE Press, 1997. 115

[24] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994. Special issue for LICS 92. 115, 116, 126

[25] Y. Kesten, Z. Manna, and A. Pnueli. Verifying clocked transition systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, pages 13–40. Springer-Verlag, 1996. 115

[26] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998. 119

[27] S. Mukhopadhyay and A. Podelski. Beyond region graphs: Symbolic forward analysis of timed automata. In C. Pandurangan, V. Raman, and R. Ramanujam, editors, *19th International Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *LNCS*, pages 233–245, December 1999. 115

[28] S. Mukhopadhyay and A. Podelski. Model checking for timed logic processes. In *Proceedings of the First International Conference on Computational Logic (CL)*, volume 1861 of *LNAI*, pages 598–612. Springer, 2000. 115

[29] U. Nilsson and J. Lübke. Constraint logic programming for local and symbolic model checking. In *Proceedings of the First International Conference on Computational Logic (CL)*, volume 1861 of *LNAI*, pages 384–398. Springer, 2000. 126

[30] G. Pemmasani, C. Ramakrishnan, and I. Ramakrishnan. Efficient model checking of real time systems using tabled logic programming and constraints. In *This proceedings*, 2002. 126

[31] T. Przymusinski. On the semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988. 119

[32] Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*. Springer-Verlag, June 1997. 126

[33] H. Seki and H. Ito. A query evaluation method for stratified programs under the extended cwa. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 195–211. MIT Press, 1988. 121, 126

[34] P. Stuckey. Negation for constraint logic programming. *Information and Computation*, 118(1), 1995. 126

[35] P. J. Stuckey. Constructive negation for constraint logic programming. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 328–339. IEEE Computer Society Press, 1991. 124

[36] H. Tamaki and T. Sato. OLD resolution with tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, volume 225 of *LNCS*, pages 84–98, London, 1986. Springer-Verlag. 121, 126

[37] D. Toman. Top-down beats bottom-up for constraint extensions of datalog. In J. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 98–114, Cambridge, Dec. 4–7 1995. MIT Press. 126

[38] S. Tripakis. Verifying progress in timed systems. In J. P. Katoen, editor, *Formal Methods in Real time and Probablistic Systems: 5th International AMAST Workshop (ARTS99)*, volume 1601 of *LNCS*, pages 299–314. Springer-Verlag, 1999. 115, 116, 126

A Model Theoretic Semantics for Multi-level Secure Deductive Databases*

Hasan M. Jamil and Gillian Dobbie**

[1] Department of Computer Science, Mississippi State University
jamil@cs.msstate.edu
[2] Department of Computer Science, University of Auckland
gill@cs.auckland.ac.nz

Abstract. The impetus for our current research is the need to provide an adequate framework for *belief reasoning* in multi-level secure (MLS) databases. We demonstrate that a prudent application of the concept of *inheritance* in a deductive database setting will help capture the notion of *declarative belief* and belief reasoning in MLS databases in an elegant way. In this paper, we show that these concepts can be captured in a F-logic style declarative query language, called *MultiLog*, for MLS deductive databases for which a model theoretic semantics exists. This development is significant from a database perspective as it now enables us to compute the semantics of MultiLog databases in a bottom-up fashion. The semantics developed here is reminiscent of the stable model semantics of logic programs with negation. We also define a bottom-up procedure to compute unique models of stratified MultiLog databases. Finally, we also establish the equivalence of MultiLog's three logical characterizations – model theory, fixpoint theory and proof theory.

Keywords: MLS databases, security, belief assertion, reasoning, deductive databases, soundness and completeness.

1 Introduction

The multi-level secure (MLS) data model was proposed to overcome the limitations of the traditional authorization scheme that required defining complicated views on a per user basis that essentially limits access to an entire set of columns of a relation in an *all* or *nothing* fashion. Under this protocol, authorization at the individual data level could not be defined easily. It was shown that a more flexible and fine grain, yet effective, authorization protocol for increased sharing of knowledge could be easily provided by the MLS data model [3,6,13]. Hence, several algebra and SQL based languages were developed to manipulate and query MLS databases. Recently, at least two logic based query languages for MLS databases were proposed [2,8] to extend the functionalities of relational

* Research supported in part by a grant from the National Science Foundation.
** Research completed while the author was visiting Mississippi State University in October 2000, while she was on sabbatical at the National University of Singapore.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 130–147, 2002.
© Springer-Verlag Berlin Heidelberg 2002

MLS languages and to overcome their proven limitations. The needs and merits of a deductive metaphor of the MLS model is eloquently discussed in [16]. However, the paucity of attempts aimed at developing a logical characterization for MLS models suggests that MLS deductive databases are really at their embryonic stage. While several proposals addressed the general issue of authorization in a deductive framework [11,1,5,15], to our knowledge, only Cuppens [2] and Jamil [8] addressed the issue of querying MLS deductive databases.

The impetus for MultiLog was to (i) develop a model and a query language which can guard against an additional security breach, called the surprise stories [8], (ii) provide linguistic instruments for capturing multiple belief modes, (iii) allow reasoning and recursion, (iv) support user defined belief modes making it possible to tailor the user view as needed, and finally (iv) enable belief speculation and belief reasoning through ad hoc belief querying. MultiLog's syntax is reminiscent of F-logic [12] but has several differences. The need for these extensions as well as the shortcomings of the MLS data model have been established in [2,10,8]. For the sake of conciseness we do not discuss the basic MLS data model in this paper. Interested readers may refer to [6] for an introductory reading.

Our goals for this paper are two-fold: (i) to develop a direct Herbrand semantics for a definite Horn clause fragment of MultiLog by defining a model theory and a fixpoint theory, and (ii) to establish equivalence of its three characterizations – proof theory (discussed in [8]), model theory and fixpoint theory. We make a critical observation that user views of the MLS databases at different security levels closely resemble the notion of *inheritance* in object oriented systems albeit in a slightly elaborate fashion. We utilize this connection and exploit our experience in dealing with inheritance in logic based systems [7,4]. The so called *belief function* introduced in [8], which we adapt in this paper for defining the notion of belief in Herbrand sets, is an extension of the inheritability function discussed in [7]. Through the equivalence of the different logical characterizations, we establish the fact that MultiLog's unique features and modeling capabilities, some of which are non-monotonic in nature, do not compromise the soundness and completeness of the language. This development is significant from a theoretical perspective, as it gives insight into the understanding of the logical behavior and mathematical foundations of the language.

The paper is organized as follows. An overview of the syntax is presented in section 2. Then the declarative semantics is given in section 3 by presenting an Herbrand semantics in section 3.2, and a corresponding fixpoint semantics in section 3.3. The logical equivalence of all three characterizations is established in section 3.3 before we conclude in section 4. We have omitted all proofs due to lack of space. The proofs can be found in the technical report in [9].

2 Overview of MultiLog Language

The language $\mathcal{L}$ of MultiLog is a 7-tuple $\langle \mathcal{P}, \mathcal{F}, \boldsymbol{A}, \mathcal{V}, \mathcal{S}, \preceq, \mu \rangle$ where (i) $\mathcal{P}$ is an infinite set of predicate names, (ii) $\mathcal{F}$ is an infinite set of constants including

the symbol `null`, denoted $\perp$, (iii) $\boldsymbol{A}$ is a finite set of attribute names, (iv) $\mathcal{V}$ is a denumerable set of variable names, (v) $\mathcal{S}$ is a finite non-empty set of labels which includes a special symbol $s_\perp$ intended to denote the security levels in our language, (vi) $\preceq$ is a partial order on the symbols in $\mathcal{S}$ that captures the idea of the hierarchy of security levels, and finally (for this paper) (vii) $\mu = \{cau, opt, fir\}$ is a finite set of symbols for belief modes. The symbols in $\mathcal{L}$ are pairwise disjoint. The terms $\mathcal{T}$ of $\mathcal{L}$ are the set $\mathcal{F} \cup \mathcal{V}$.

Formulas and Databases: There are five types of atoms in our language: m-, b-, p-, l- and h-atoms[1].

- MLS atoms or m-atoms: Let p be a predicate symbol in $\mathcal{P}$ of arity n - denoted p/n, a is an attribute name in $\boldsymbol{A}$, $\boldsymbol{v}$ and $\boldsymbol{k}$ are terms in $\mathcal{T}$, and $\boldsymbol{s}$ and $\boldsymbol{c}$ are symbols in $\mathcal{S} \cup \mathcal{V}$. Then $\boldsymbol{s}[p(\boldsymbol{k} : a \xrightarrow{\boldsymbol{c}} \boldsymbol{v})]$ is an m-atom. Intuitively, an m-atom represents an attribute of a tuple as in an MLS relational database counterpart where a is an attribute name, $\boldsymbol{v}$ is a value, $\boldsymbol{k}$ is the object key and $\boldsymbol{s}$ and $\boldsymbol{c}$ are security labels. The label $\boldsymbol{s}$ denotes the security level of the predicate p and mimics the tuple classification TC in the MLS relational model. The label $\boldsymbol{c}$ mimics the attribute classification in the MLS relational model.
- Believed atoms or b-atoms: Let $\boldsymbol{s}[p(\boldsymbol{k} : a \xrightarrow{\boldsymbol{c}} \boldsymbol{v})]$ be an m-atom and $m \in \mu$ be a mode of belief by a rational agent. Then $\boldsymbol{s}[p(\boldsymbol{k} : a \xrightarrow{\boldsymbol{c}} \boldsymbol{v})] \ll m$ is a b-atom. Intuitively, a b-atom says that a rational agent believes $p(\boldsymbol{k} : a \xrightarrow{\boldsymbol{c}} \boldsymbol{v})$ at level $\boldsymbol{s}$ in a mode m.
- Predicates or p-atoms: If p is a predicate symbol in $\mathcal{P}$ of arity k - denoted p/k, and $\boldsymbol{t}_1, \ldots, \boldsymbol{t}_k$ are terms in $\mathcal{T}$, then $p(\boldsymbol{t}_1, \ldots, \boldsymbol{t}_k)$ is a p-atom. The sense of p-atoms is identical to predicates in classical logic.
- Level atoms or l-atoms: Let $level/1$ be a distinguished predicate symbol in $\mathcal{P}$, and $\boldsymbol{s}$ be a symbol in $\mathcal{S} \cup \mathcal{V}$. Then $level(\boldsymbol{s})$ is an l-atom. An l-atom declares the existence of a security level in a database Δ.
- Hierarchy atoms or h-atoms: Let $order/2$ be another distinguished predicate symbol in $\mathcal{P}$, and $\boldsymbol{l}$ and $\boldsymbol{h}$ be two symbols in $\mathcal{S} \cup \mathcal{V}$. Then $order(\boldsymbol{l}, \boldsymbol{h})$ is an h-atom. Intuitively an h-atom asserts that the security level $\boldsymbol{l}$ is lower than $\boldsymbol{h}$ and that there are no other $\boldsymbol{i}$ such that $order(\boldsymbol{l}, \boldsymbol{i})$ and $order(\boldsymbol{i}, \boldsymbol{h})$ hold.

Formulas of $\mathcal{L}$ are defined as usual. A literal is either an atom ($\mathcal{A}$) or the negation of an atom ($\neg\mathcal{A}$). Following the custom in logic programming, we only consider the definite (Horn) clause fragment of our language. A clause in $\mathcal{L}$ is an expression of the form $\mathcal{A} \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$ such that $\mathcal{A}$ and $\mathcal{B}_i$s are atoms of $\mathcal{L}$. If the consequent of a clause is an m-atom, we call the clause an m-clause. Similarly, we define p-, l- and h-clauses. We, however, do not have b-clauses as we do not allow b-atoms to appear in the consequent.

[1] Throughout the paper we use uppercase letters for variables and lowercase letters for constants. We use bold lowercase letters to denote arbitrary terms. While caligraphic uppercase letters are used to denote arbitrary formulas, uppercase italic letters A, B, F, G, etc. are used to denote ground formulas.

Tid	**Starship**	C_1	**Objective**	C_2	**Destination**	C_3	**TC**
t_1	Voyager	U	Spying	S	Mars	U	S
t_2	Phantom	U	Spying	S	Omega	U	S
t_3	Phantom	C	Supply	C	Venus	S	S
t_4	Atlantis	U	Diplomacy	U	Vulcan	U	C
t_5	Voyager	U	Training	U	Mars	U	U

Fig. 1. MLS relation *Mission*(Starship, C_1, Objective, C_2, Destination, C_3, TC)

Example 2.1. Consider the MLS relation Mission*(Starship, C_1, Objective, C_2, Destination, C_3, TC) in Figure 1 adapted from [8]. In this example, and also throughout this paper, we consider only four security levels for simplicity. Namely, $s_\perp$, u, c, and s with a total order $s_\perp < u < c < s$. That is, we have in our database the atoms* order($s_\perp$,u), order(u,c), *and* order(c,s). *The attributes C_1, C_2, and C_3 respectively denote the attribute classifications for Starship, Objective and Destination, whereas TC denotes the tuple classification for* Mission.

In this scheme, Starship is the apparent primary key *(AK), C_1 is the classification of the apparent primary key (C_{AK}), and for the attributes Objective and Destination respectively, the functional dependencies {Starship, C_{AK}, C_2} → Objective, and {Starship, C_{AK}, C_3} → Destination hold where C_{AK}=C_1. Hence, the* primary key *of the relation is {Starship,C_{AK},C_2,C_3}={Starship,C_1,C_2,C_3}.*

Now consider tuple t_2 in Mission*. It is easy to see that a predicate representation of t_2 would be $mission(phantom, u, spying, s, omega, u, s)$. In MultiLog we represent t_2 as $s[mission(phantom : starship \xrightarrow{u} phantom)] \wedge s[mission(phantom : objective \xrightarrow{s} spying)] \wedge s[mission(phantom : destination \xrightarrow{u} omega)]$, i.e., as a conjunction of m-atoms of the form* TC[rel(app_key:att_name $\xrightarrow{att_class}$ att_val)]. *A more compact representation is an* m-molecule, *$s[mission(phantom : starship \xrightarrow{u} phantom, objective \xrightarrow{s} spying, destination \xrightarrow{u} omega)]$. An m-molecule is a syntactic variant of conjunctions of m-atoms. In general, for any classical predicate representation $p(\boldsymbol{k}, a_1, \boldsymbol{v}_1, \boldsymbol{c}_1, \ldots, a_k, \boldsymbol{v}_k, \boldsymbol{c}_k, \boldsymbol{s})$ corresponding to an MLS tuple over the scheme $p(K, a_1, C_1, \ldots, a_k, C_k, TC)$ where $K \subseteq \{a_1, \ldots, a_k\}$ is the apparent primary key, the m-molecular representation can be written as $\boldsymbol{s}[p(\boldsymbol{k} : a_1 \xrightarrow{\boldsymbol{c}_1} \boldsymbol{v}_1, \ldots, a_k \xrightarrow{\boldsymbol{c}_n} \boldsymbol{v}_n)]$. Notice that, ordinarily an MLS tuple will be represented in classical first-order logic as a predicate of the form $p(\boldsymbol{v}_1, \boldsymbol{c}_1, \ldots, \boldsymbol{v}_k, \boldsymbol{c}_k, \boldsymbol{s})$. The only difference with MLS tuples is, of course, that we include attribute names a_is and the apparent key $\boldsymbol{k}$ in our atoms/molecules as just shown. A similar approach was also taken in [12].* □

Definition 2.1 (Depth of Atoms). Let $\mathcal{A}$ be an atom in $\mathcal{L}$. If $\mathcal{A}$ is p-, l- or h-atom then the depth of $\mathcal{A}$, written $depth(\mathcal{A})$ is $s_\perp$. It is $\boldsymbol{l}$[2], otherwise (i.e., m- or b-atoms).

[2] We are assuming that the atom $\mathcal{A}$ is of the form $\boldsymbol{l}[p(\boldsymbol{k} : a \xrightarrow{\boldsymbol{c}} \boldsymbol{v})]$ or $\boldsymbol{l}[p(\boldsymbol{k} : a \xrightarrow{\boldsymbol{c}} \boldsymbol{v})] \ll m$.

Intuitively, $s_\perp$ has the lowest security classification. This is a technical requirement and guarantees that all the l-, h- and p-atoms defined in $s_\perp$ are visible to all the higher levels.

Definition 2.2 (Dependency Graph). Let Cl be a clause of the form $\mathcal{A} \leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$. Let $\leftharpoonup$ denote the binary relationship *depends on*. For Cl, we say that $\mathcal{A}$ depends on $\mathcal{B}_1, \ldots, \mathcal{B}_m$, denoted $\mathcal{A} \leftharpoonup \mathcal{B}_1, \ldots, \mathcal{A} \leftharpoonup \mathcal{B}_m$. The transitive closure of the relation $\leftharpoonup$ with respect to $\mathcal{A}$ is called the *dependency graph* of $\mathcal{A}$. □

Definition 2.3 (Databases and Queries). A *database* Δ, or equivalently a program $\mathbf{P}$, inMultiLog is an expression of the form $\langle \Lambda, \Sigma, \Pi, \mathcal{Q} \rangle$, where (i) Λ is a set of l- and h-clauses (possibly empty) defining the security levels and inducing a partial order on the security levels (symbols in $\mathcal{S} \cup \mathcal{V}$), (ii) Σ is a set of m-clauses defining the secured data component of Δ, (iii) Π is a set of p-clauses (possibly empty), and finally (iv) $\mathcal{Q}$ is a set of clauses of the form $\leftarrow \mathcal{B}_1, \ldots, \mathcal{B}_m$, called the queries. For every clause $Cl = \mathcal{A} \leftarrow \mathcal{G} \in \Lambda$, we require that the dependency graph of $\mathcal{A}$ does not contain atoms other than h- or l-atoms[3]. □

Similar to MLS relational models, we require that MultiLog database m-molecules satisfy several integrity constraints. First, we require that for every m-molecule there is a key attribute AK for which the value is $\boldsymbol{k}$. Hence, there must be an m-atom of the form $\boldsymbol{s}[p(\boldsymbol{k} : a \xrightarrow{\boldsymbol{c}} \boldsymbol{k})]$. That is, for every m-atom of the form $\boldsymbol{s}[p(\boldsymbol{k} : b \xrightarrow{\boldsymbol{d}} \boldsymbol{v})]$ in a database P[4], we also have $\boldsymbol{s}[p(\boldsymbol{k} : a \xrightarrow{\boldsymbol{c}} \boldsymbol{k})]$. For such atoms, $\boldsymbol{k}$ is identified as AK, $\boldsymbol{c}$ is as $\boldsymbol{c}_{AK}$, $\boldsymbol{s}$ as TC, and for all other atoms for which $\boldsymbol{k}$ is the key, a is identified as A_i, $\boldsymbol{c}$ as C_i and $\boldsymbol{v}$ as A_i in a manner similar to Jajodia and Sandhu [6].

Definition 2.4 (Closure of Databases). Let $\Delta = \langle \Lambda, \Sigma, \Pi, \mathcal{Q} \rangle$ be a MultiLog database. The *closure* $\Delta^* = \langle \Lambda^*, \Sigma^*, \Pi^*, \mathcal{Q}^* \rangle$ of Δ is the smallest set of clauses satisfying the conditions below:

- $\Lambda \subseteq \Lambda^*, \Sigma^* = \Sigma, \Pi^* = \Pi, \mathcal{Q}^* = \mathcal{Q}$.
- $L \preceq L \leftarrow level(L) \in \Lambda^*$.
- $L \preceq L \leftarrow order(L, H) \in \Lambda^*$ and $H \preceq H \leftarrow order(L, H) \in \Lambda^*$.
- $L \preceq H \leftarrow L \preceq H', order(H', H) \in \Lambda^*$. □

In the remainder of this paper, unless stated otherwise, we consider only closed databases.

Definition 2.5 (Level of Databases). Let Δ be a consistent database, and u is a user with a clearance $\bar{u}$. The database Δ is in level $\bar{u}$, denoted $\langle \Delta, \bar{u} \rangle$, if the user u with clearance $\bar{u}$ accesses Δ. □

[3] Intuitively, it means that the ground closure of Λ does not depend on the clauses defined in other components of Δ.

[4] In fact in the model of P defined shortly.

Example 2.2. The Mission relation in Figure 1 is expressed in MultiLog as the level s database $\langle mission, s\rangle$ below.

$$l\Lambda_{D_1} := \Big| level(u).\ level(c).\ level(s).\ order(u,c).\ order(c,s).$$

$$\Sigma_{D_1} := \left| \begin{array}{l} u[mission(voyager : starship \xrightarrow{u} voyager, objective \xrightarrow{u} \\ \qquad training, destination \xrightarrow{u} mars)]. \\ c[mission(atlantis : starship \xrightarrow{u} atlantis, objective \xrightarrow{u} \\ \qquad diplomacy, destination \xrightarrow{u} vulcan)]. \\ s[mission(phantom : starship \xrightarrow{c} phantom, objective \xrightarrow{c} \\ \qquad supply, destination \xrightarrow{s} venus)]. \\ s[mission(phantom : starship \xrightarrow{u} phantom, objective \xrightarrow{s} \\ \qquad spying, destination \xrightarrow{u} omega)]. \\ s[mission(voyager : starship \xrightarrow{u} voyager, objective \xrightarrow{s} \\ \qquad spying, destination \xrightarrow{u} mars)]. \end{array} \right.$$

Belief modes give the user the choice to reason and theorize about the beliefs. In this paper, we describe three belief modes, and it is possible for users to define further belief modes. We now informally introduce the three belief modes.

If the user takes the firm view of the data, they believe the data visible at their security level only is correct and believable data. Consider the query $\leftarrow s[mission(V : W \xrightarrow{X} Y)] \ll fir$ that asks for the firm s level view of mission. The answer returned is as follows:

$$\begin{array}{r} lcls[mission(phantom : starship \xrightarrow{c} phantom, objective \xrightarrow{c} \\ \qquad supply, destination \xrightarrow{s} venus)]. \\ s[mission(phantom : starship \xrightarrow{u} phantom, objective \xrightarrow{s} \\ \qquad spying, destination \xrightarrow{u} omega)]. \\ s[mission(voyager : starship \xrightarrow{u} voyager, objective \xrightarrow{s} \\ \qquad spying, destination \xrightarrow{u} mars)]. \end{array}$$

If the user takes the optimisitic view, they believe what is created at all security levels is correct and believable data. Hence the following is the response if we consider the query $\leftarrow s[mission(V : W \xrightarrow{X} Y)] \ll opt$ that asks for the optimistic s level view of mission:

$$\begin{array}{r} lcls[mission(voyager : starship \xrightarrow{u} voyager, objective \xrightarrow{u} \\ \qquad training, destination \xrightarrow{u} mars)]. \\ s[mission(atlantis : starship \xrightarrow{u} atlantis, objective \xrightarrow{u} \\ \qquad diplomacy, destination \xrightarrow{u} vulcan)]. \\ s[mission(phantom : starship \xrightarrow{c} phantom, objective \xrightarrow{c} \end{array}$$

$$\begin{aligned} & supply, destination \xrightarrow{s} venus)]. \\ s[mission(phantom : & starship \xrightarrow{u} phantom, objective \xrightarrow{s} \\ & spying, destination \xrightarrow{u} omega)]. \\ s[mission(voyager : & starship \xrightarrow{u} voyager, objective \xrightarrow{s} \\ & spying, destination \xrightarrow{u} mars)]. \end{aligned}$$

If the user takes the cautious view, the visible information at a given level that has the highest security classification is retained and the other data is filtered out. The assumption is that the higher level information is more reliable and takes priority over the lower level information. So, the query $\leftarrow s[mission(V : W \xrightarrow{X} Y)] \ll cau$ *that asks for the cautious s level view of mission returns:*

$$\begin{aligned} lcls[mission(atlantis : & starship \xrightarrow{u} atlantis, objective \xrightarrow{u} \\ & diplomacy, destination \xrightarrow{u} vulcan)]. \\ s[mission(phantom : & starship \xrightarrow{c} phantom, objective \xrightarrow{s} \\ & spying, destination \xrightarrow{s} venus)]. \\ s[mission(voyager : & starship \xrightarrow{u} voyager, objective \xrightarrow{s} \\ & spying, destination \xrightarrow{u} mars)]. \end{aligned}$$

In this example, we have demonstrated how belief atoms are used in queries and the results of the different belief modes. The use of belief atoms in the right hand side of a rule follows from this. □

3 Declarative Semantics of MultiLog

The Herbrand semantics of MultiLog databases is defined in terms of a composite set-theoretic structure which provides a model for each of the security levels in $\mathcal{L}$ (represented by each of the symbols in $\mathcal{S}$), including the level $s_\perp$ – the system level which is not part of any database universe. In other words, each model in the composite structure interprets formulas pertaining to the corresponding levels in the security hierarchy. The notion of *"belief"* in such a structure is then captured using a function level semantics over the sets in the Herbrand structure, not as a set membership, as explained shortly.

These structures are similar to standard interpretations and models in classical logic. The difference between the structures for standard predicate logic programs and ours is that while we simultaneously interpret several program fragments that are interrelated through *inter structure entailment* relations in the composite structure (similar to message passing in object oriented systems), in the predicate logic case we interpret a single program with only standard entailment due to implication and conjunction. Unlike Miller's Kripke-like structures [14], we are then able to define the notion of satisfaction in a standard way as a "set membership".

The notion of Herbrand universe $\mathcal{U}$ and base $\mathcal{H}$[5] is defined similar to the classical case. Formally, an *Herbrand structure* $\mathbf{H}$ of $\mathcal{L}$ is a tuple $\langle H(s) : s \in \mathcal{S} \rangle$ such that $H(s) \subseteq \mathcal{H}$ for every $s \in \mathcal{S}$. When $s \not= s_\perp$, $H(s)$ contains only m-atoms, otherwise $H(s_\perp)$ contains only p-, l- and h-atoms. Intuitively, every $H(s), s \not= s_\perp$ in $\mathbf{H}$ interprets the associated data items belonging to the level s as ground m-atoms that are true with respect to level s in $\mathbf{H}$. To make a distinction between an interpretation corresponding to a security level and the interpretation structures for our language $\mathcal{L}$, we henceforth call them interpretations and T-interpretations respectively, since the latter is actually a tuple of simple interpretations or sets.

3.1 Formula Satisfaction and Models

Before we proceed to define the notion of truth and satisfaction in our language, we need to cast the belief function β defined in [8] in terms of sets of ground formulas in MultiLog.

Definition 3.1 (Belief in Herbrand Sets). Let S be an arbitrary set of m-atoms, $A = l[p(k : a \xrightarrow{c} v)]$ be a ground m-atom, and m be a belief mode in μ={*fir, cau, opt*}. Then, the belief function $\beta : \mathcal{P}(\mathbf{S}) \times \mathcal{S} \times \mu \rightarrow \mathcal{P}(\mathbf{S})$, where $\mathbf{S}$ is a set of all possible m-atoms, is defined as follows:

$$\beta(S, l, m) = \left\{ l[p(k : a \xrightarrow{c} v)] \;\middle|\; \begin{array}{l} \text{One of the following conditions hold:} \\ \\ \text{- } m = \mathit{fir} \text{ and } l[p(k : a \xrightarrow{c} v)] \in S \\ \text{- } m = \mathit{cau} \text{ and } \exists l'[p(k : a \xrightarrow{c} v)] \in S,\ l' \preceq l, \\ \quad \text{and } \neg\exists l''[p(k : a \xrightarrow{c'} v')] \in S,\ l'' \preceq l, \text{ and } c \prec c'. \\ \text{- } m = \mathit{opt} \text{ and } l'[p(k : a \xrightarrow{c} v)] \in S \text{ and } l' \preceq l. \end{array} \right. \quad \square$$

3.2 Herbrand Interpretations

We now define satisfaction of formulas in an Herbrand structure in a manner similar to the classical case. The major difference is that we now have to define satisfaction for formulas with respect to a T-interpretation (across a set of interpretations) and a user level. The definition of satisfaction must then correctly capture the meaning of secured data at a given level and the belief asserted by such data along the security hierarchy as intended by the m-atoms and b-atoms.

Definition 3.2 (Satisfaction of Formulas). Let $\mathbf{H}$ be a T-interpretation, $\bar{u}$ be a user clearance level, $H(i)$ be any arbitrary interpretation[6] in $\mathbf{H}$ where i is

[5] Note that b-atoms are not part of the Herbrand base $\mathcal{H}$ as their satisfaction really depends on the interpretations and an entailment function β. But note that the data components of b-atoms, the corresponding m-atoms, are.

[6] Note again that an interpretation is a Herbrand set, and a T-interpretation is a tuple of such Herbrand sets.

a security level in **H**, and let n be the number of such security levels[7]. Let A and B denote ground atomic formulas, and F denote any arbitrary ground formula. Then, the satisfaction of ground formulas with respect to $H(i)$, denoted $H(i)\models_{\mathbf{H},\bar{u}}F$, is defined as follows:

(1) $H(s)\models_{\mathbf{H},\bar{u}}A \iff A \in H(s)$ where $depth(A) = s$ and $s \preceq \bar{u}$
(2) $H(s)\models_{\mathbf{H},\bar{u}}A \iff H(o)\models_{\mathbf{H},\bar{u}}A$ where $depth(A) = o$ and $s \not= o$
(3) $H(i)\models_{\mathbf{H},\bar{u}}A \leftarrow B_1,\ldots,B_m \iff H(i)\models_{\mathbf{H},\bar{u}}B_g,\ g = 1,\ldots,m \Longrightarrow H(i)\models_{\mathbf{H},\bar{u}}A$
(4) $H(i)\models_{\mathbf{H},\bar{u}}l[p(k : a \xrightarrow{c} v] \ll m \iff H'\models_{\mathbf{H},\bar{u}}l[p(k : a \xrightarrow{c} v]$ where
$H' = \beta(\bigcup_{j=1}^{n} H(s_j), \bar{u}, m)$ such that
$\forall j, s_j \preceq l$ and $l \preceq \bar{u}$

Finally, we say that $\mathbf{H} \models_{\bar{u}} A$ if and only if $H(l)\models_{\mathbf{H},\bar{u}}A$, where $l = depth(A)$. □

There are several important observations that one can make in the way satisfaction is defined in our Herbrand structures. First of all, every interpretation in **H** assigns meaning to data items at a given security level. The interpretation is indexed by the label returned by the *depth* function in definition 2.1. The satisfiability of a formula does not depend *only* upon the membership of the formula in an interpretation as in its classical logic counterpart. In addition, satisfiability requires a formula to be visible, i.e., the visibility at a clearance level, which is a parameter in the definition of satisfaction. The length of the T-interpretation depends on the set of symbols in $\mathcal{S}$ in a given database. Note that if $\mathcal{S}$ contains only $s_\perp$, the T-interpretation essentially degenerates into a classical Herbrand interpretation as $\mathcal{S}$ is non-empty and always includes $s_\perp$.

The way in which we have defined our interpretations and satisfaction, rules out the possibility of having interpretation $H(s)$ satisfying a formula of the form $l[p(k : a \xrightarrow{c} v)]$ such that $l \not= s$. But it may still satisfy, for example, a formula of the form $s[p(k : a \xrightarrow{c} v)]$ such that $s \preceq c$. We thus require our candidate interpretations to satisfy additional consistency requirements as defined in [8] to be a model of a database. Note that these conditions are reminiscent and adaptation of the core integrity properties defined in Jajodia and Sandhu [6] with some significant differences. In particular, notice that except for the entity, null and polyinstantiation integrity constraints (core integrity properties of MLS relational model [6]), the closure and hierarchy integrity are the added requirements.

Recall that, using the cautious belief mode, if a fact isn't believed at a particular security level, then a fact at a lower level is believed. This concept of specificity is intimately related to the notion of inheritance and overriding in object-oriented systems. We attempt to clarify this through the example below and the definition that follows formalizes the idea of the so called *more specific* atoms.

Example 3.1. Consider the level s database $\langle D_1, s\rangle$ *shown below.*

[7] Notice that in this definition the symbols s, i, o, u, l and s_j, as well as $\bar{u}$, represent security levels in **H**.

$$\Lambda_{D_1} := \left| \begin{array}{l} r_1 : level(u). \\ r_2 : level(c). \\ r_3 : level(s). \\ r_4 : order(u, c). \\ r_5 : order(c, s). \end{array} \right. \qquad \Sigma_{D_1} := \left| \begin{array}{l} r_6 : c[p(k : a \xrightarrow{u} v)]. \\ r_7 : c[p(k : a \xrightarrow{c} t)] \leftarrow p(j). \\ r_8 : s[p(K : A \xrightarrow{C} V)] \leftarrow \\ \quad c[p(K : A \xrightarrow{C} V)] \ll cau. \end{array} \right.$$

$$\mathcal{Q}_{D_1} := \left| r_{10} : ?\ s[p(k : a \xrightarrow{u} v)] \right.$$

If $p(j)$ is not true, then $s[p(k : a \xrightarrow{u} v)]$ is true. But if $p(j)$ is true, then $s[p(k : a \xrightarrow{c} t)]$ is true. We say that the atom $c[p(k : a \xrightarrow{c} t)]$ is more specific than the atom $c[p(k : a \xrightarrow{u} v)]$ because $u < c$ (because of $order(u, c) \in \Lambda_{D_1}$). □

Definition 3.3 (More Specific). Let $\langle \Delta, \bar{u} \rangle$ be level $\bar{u}$ database, where $A = l[r(k : a \xrightarrow{s} v)]$ and $A' = l[r(k : a \xrightarrow{s'} v')]$, where $s' < s$. We say A is *more specific* than A'.

Example 3.2. The m-atom $c[p(k : a \xrightarrow{c} t)]$ is more specific than $c[p(k : a \xrightarrow{u} v)]$ because $u < c$. We can't say anything about the specificity of $s[p(k : a \xrightarrow{c} t)]$ and $c[p(k : a \xrightarrow{u} v)]$.

Definition 3.4 (Consistent Database). A database $\langle \Delta, \bar{u} \rangle$ is consistent if there exists a mapping μ from the set of ground atoms to the set of non-negative integers, such that, for every atom in every ground instance $C\theta$ for every clause C in $\langle \Delta, \bar{u} \rangle$,

1. $\mu(A') \leq \mu(A)$, where A' is an atom in the body of the clause instance $C\theta$, and A is the head of the clause instance $C\theta$,
2. $\mu(B) < \mu(A')$, for every ground clause $C'\theta'$ where B is a cautious b-atom in the body of $C\theta$ and B' is a cautious b-atom in the body of $C'\theta'$, B is more specific than B', and A' is the head of $C'\theta'$.
3. $\mu(A) < \mu(A')$, for every ground clause $C'\theta'$ where B is a cautious b-atom in the body of $C\theta$ and B' is a cautious b-atom in the body of $C'\theta'$, B is more specific than B', A is the head of $C\theta$, and A' is the head of $C'\theta'$.
4. $\mu(A') < \mu(A)$, for every ground clause $C'\theta'$ where B is a cautious b-atom in the body of $C\theta$, A is the head of $C\theta$, A' is the head of $C'\theta'$ and A' is more specific than B. □

Example 3.3. A database with a rule $s[p(k : a \xrightarrow{c} t)] \leftarrow s[p(k : a \xrightarrow{u} v)] \ll cau$ is not consistent. Using the definition of a consistent database, we find $s[p(k : a \xrightarrow{c} t)] < s[p(k : a \xrightarrow{c} t)]$ and there is no mapping from this to the set of non-negative integers. Intuitively, this is not a consistent database, since if the body of the rule is satisfied, then the head of the rule is satisfied, and the body is no longer satisfied.

Example 3.4. Consider the following ground level s database $\langle D_2, s \rangle$ for simplicity as a further example.

$$\Lambda_{D_2} := \left| \begin{array}{l} r_1 : level(u). \\ r_2 : level(c). \\ r_3 : level(s). \\ r_4 : order(u, c). \\ r_5 : order(c, s). \end{array} \right. \qquad \Sigma_{D_2} := \left| \begin{array}{l} r_6 : c[p(k : a \xrightarrow{u} v)]. \\ r_7 : c[p(k : a \xrightarrow{c} t)] \leftarrow p(j). \\ r_8 : s[p(k : a \xrightarrow{c} t)] \leftarrow c[p(k : a \xrightarrow{c} t)] \ll cau. \\ r_9 : s[p(k : a \xrightarrow{u} v)] \leftarrow c[p(k : a \xrightarrow{u} v)] \ll cau. \end{array} \right.$$

Based on the definition for consistent databases, we get:

$$\begin{array}{ll} p(j) & \leq c[p(k : a \xrightarrow{c} t)] \ (rule\ 1) \\ c[p(k : a \xrightarrow{c} t)] & \leq s[p(k : a \xrightarrow{c} t)] \ (rule\ 1) \\ c[p(k : a \xrightarrow{u} v)] & \leq s[p(k : a \xrightarrow{u} v)] \ (rule\ 1) \\ c[p(k : a \xrightarrow{c} t)] & < s[p(k : a \xrightarrow{u} v)] \ (rule\ 2) \\ s[p(k : a \xrightarrow{c} t)] & < s[p(k : a \xrightarrow{u} v)] \ (rules\ 3\ and\ 4) \end{array}$$

This goes to show that there exists a mapping from the set of ground atoms to the set of non-negative integers, so database $\langle D_2, s\rangle$ is consistent.

Consider if the above database is altered in the following way: rules 8 and 9 are removed and replaced by $s[p(k : a \xrightarrow{c} t)] \leftarrow s[p(k : a \xrightarrow{u} v)] \ll cau$, then by rule 4, $s[p(k : a \xrightarrow{c} t)] < s[p(k : a \xrightarrow{c} t)]$ and there is no mapping from the set of ground atoms to the set of non-negative integers, so the database is not consistent. Intuitively, what does this mean? If the body of the clause is true, then the head of the clause becomes true and as a consequence the body of the rule is no longer true. This database has no meaning. □

We now define the notion of a model for databases in MultiLog based on the notion of satisfaction and consistent interpretations.

Definition 3.5 (Consistent T-models). Let $\mathbf{H} = \langle H(s) : s \in \mathcal{S}\rangle$ be a Herbrand structure, or a T-interpretation, $\langle \Delta, \bar{u}\rangle$ be a database at level $\bar{u}$, and $|\langle \Delta, \bar{u}\rangle|$ be the Herbrand instantiation of $\langle \Delta, \bar{u}\rangle$. Then $\mathbf{H}$ is a T-model for $\langle \Delta, \bar{u}\rangle$ iff for every clause cl in $|\langle \Delta, \bar{u}\rangle|$, $\mathbf{H} \models_{\bar{u}} cl$. $\mathbf{H}$ is a consistent T-model for $\langle \Delta, \bar{u}\rangle$ iff $\mathbf{H}$ is a T-model for $\langle \Delta, \bar{u}\rangle$ and is consistent as a structure. □

Example 3.5. Consider the database D_1 in example 3.1, this time at level c (as opposed to level s as shown before), i.e., $\langle D_1, c\rangle$. For the database $\langle D_1, c\rangle$, let the T-model be M_1, as shown below:

$$M_1 = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c),\ level(s), \\ order(u, c),\ order(c, s), p(j) \end{array} \right\}}_{M_1(s_\perp)}, \underbrace{\emptyset}_{M_1(u)}, \underbrace{\left\{ \begin{array}{l} \{c[p(k : a \xrightarrow{u} v)], \\ c[p(k : a \xrightarrow{c} t)] \end{array} \right\}}_{M_1(c)}, \underbrace{\{s[p(k : a \xrightarrow{c} t)]\}}_{M_1(s)} \rangle$$

In the T-model M_1 above, the first set in the interpretation belongs to level $s_\perp$, i.e., $M_1(s_\perp)$. The second set belongs to u, the third to level c, and the last to s. Now, several observations can be made here. Note that the database is at level c.

Also note that $s[p(k:a \xrightarrow{c} t)] \in M_1$, $M_1(s)$ *to be precise. Still* $M_1 \not\models\!\!\mid_c s[p(k:a \xrightarrow{c} t)]$. *This is because* $M_1(s) \not\models\!\!\mid_{M_1,c} s[p(k:a \xrightarrow{c} t)]$ *as it does not satisfy Condition 1 in the definition of satisfaction 3.2, i.e.,* $s \not/\preceq$. *But* $M_1 \models_c c[p(k:a \xrightarrow{u} v)]$, *and also* $M_1 \models_c c[p(k:a \xrightarrow{c} t)]$. *Yet, it is interesting to verify that* $M_1 \models_c c[p(k:a \xrightarrow{c} t)] \ll cau$ *but* $M_1 \not\models\!\!\mid_c c[p(k:a \xrightarrow{u} v)] \ll cau$. *This observation follows from the definition of cautious belief in Herbrand sets, i.e., definition 3.1.* □

But not every consistent T-model is *"intended"*, a term which will be defined shortly. Recall that the satisfaction of b-atoms depends on the belief function β which makes use of the Herbrand sets in **H**. Also recall that the set computed by β depends on the elements in the Herbrand sets corresponding to security levels dominated by $\bar{u}$. While the satisfaction of b-atoms is not affected by elements not required for a structure to be a T-model for a database $\langle \Delta, \bar{u} \rangle$, it potentially affects the beliefs of users as unwanted models may result. The following example helps clarify this point.

Example 3.6. Consider a level s database $\langle D_3, s \rangle$. *Assume that database* D_3 *is derived from database* D_1 *of example 3.1 by replacing rule* r_6 *with* $u[p(k:a \xrightarrow{u} v)]$, *rule* r_8 *by* $s[p(k:a \xrightarrow{u} v)] \leftarrow c[p(k:a \xrightarrow{u} v)] \ll cau$, *and finally by deleting rule* r_9 *and adding two rules* $r_{11} : p(X) \leftarrow r(X)$ *and* $r_{12} : r(X) \leftarrow q(X)$. *Now for database* $\langle D_3, s \rangle$ *as defined, the intended model* M_3 *may be identified as follows:*

$$M_3 = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c),\ level(s), \\ order(u,c),\ order(c,s) \end{array} \right\}}_{M_3(s_\perp)}, \underbrace{\{u[p(k:a \xrightarrow{u} v)]\}}_{M_3(u)}, \underbrace{\emptyset}_{M_3(c)}, \underbrace{\{s[p(k:a \xrightarrow{u} v)]\}}_{M_3(s)} \rangle$$

However, it is easy to verify that M_3' *or* M_3'' *below are not intended although they are T-models for* D_3 *at level s.*

$$M_3' = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c),\ level(s), \\ order(u,c),\ order(c,s) \end{array} \right\}}_{M_3'(s_\perp)}, \underbrace{\{u[p(k:a \xrightarrow{u} v)]\}}_{M_3'(u)}, \underbrace{\{c[p(k:a \xrightarrow{c} t)]\}}_{M_3'(c)}, \underbrace{\emptyset}_{M_3'(s)} \rangle$$

$$M_3'' = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c),\ level(s), \\ order(u,c),\ order(c,s), p(j) \end{array} \right\}}_{M_3''(s_\perp)}, \underbrace{\{u[p(k:a \xrightarrow{u} v)]\}}_{M_3''(u)}, \underbrace{\{c[p(k:a \xrightarrow{c} t)]\}}_{M_3''(c)}, \underbrace{\emptyset}_{M_3''(s)} \rangle$$

M_3' *and* M_3'' *are not intended because they make* $s[p(k:a \xrightarrow{u} v)]$ *false, i.e.,* $M_3' \not\models\!\!\mid_s s[p(k:a \xrightarrow{u} v)]$ *and* $M_3'' \not\models\!\!\mid_s s[p(k:a \xrightarrow{u} v)]$, *for similar reasons (for* $c[p(k:a \xrightarrow{c} t)]$ *being in* $M_3'(c)$ *and as well as in* $M_3''(c)$ *that made satisfaction of* $c[p(k:a \xrightarrow{u} v)] \ll cau$ *not possible, instead forced* $c[p(k:a \xrightarrow{c} t)]$ *to be cautiously believed at level c). If either one of these T-models were minimal, it*

would have modeled $s[p(k : a \xrightarrow{u} v)]$*, as dictated by logical entailment and implication. The following definition of* intended T-models *eliminates this possibility.* □

We can define a priority relationship between ground atoms, and use this to define a preference relationship between models. We prefer models in which there are fewer occurrences of higher priority atoms.

Definition 3.6 (Priority Relationship). Let $\langle \Delta, \bar{u} \rangle$ be level u database. We define a priority relationship $<_p$ on the ground atoms in $\langle \Delta, \bar{u} \rangle$. We write $a \leq_p b$ to denote $a <_p b$ or $a = b$. For every ground instance $C\theta$ of a clause in $\langle \Delta, \bar{u} \rangle$,

1. $A \leq_p B$, where A is the head of $C\theta$ and B is a ground atom in the body of $C\theta$.
2. $A <_p B'$, where A is the head of $C\theta$, B' is a cautious b-atom in the body of $C'\theta'$, B is a cautious b-atom in the body of $C\theta$, and B' is more specific than B.
3. $A \leq_p A'$, where A is the head of $C\theta$, B' is a cautious b-atom in the body of $C'\theta'$, and the head A' of $C'\theta'$ is more specific than A.

and for all ground atoms A, B, C and F:

– if $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$,
– if $A \leq_p B$ and $B <_p C$, (respectively $F <_p A$) then $A <_p C$
– if $A \leq_p B$ and $B <_p C$, (respectively $F <_p A$) then $A <_p C$ (respectively $F <_p B$).

Example 3.7. Consider the database in example 3.4, there are two ground rules $s[p(k : a \xrightarrow{u} v)] \leftarrow c[p(k : a \xrightarrow{u} v)] \ll cau$ *and* $s[p(k : a \xrightarrow{c} t)] \leftarrow c[p(k : a \xrightarrow{c} t)] \ll cau$*, and* $c[p(k : a \xrightarrow{c} v)]$ *is more specific than* $c[p(k : a \xrightarrow{u} t)]$*. Based on case (2), in the above definition* $s[p(k : a \xrightarrow{u} v)] <_p c[p(k : a \xrightarrow{c} t)]$.

Example 3.8. Again, consider the level s database $\langle D_2, s \rangle$ *in example 3.4. Based on the definition for priority relationship, we get:*

$$
\begin{aligned}
c[p(k : a \xrightarrow{c} t)] &\leq_p p(j) && (rule\ 1)\\
s[p(k : a \xrightarrow{c} t)] &\leq_p c[p(k : a \xrightarrow{c} t)] && (rule\ 1)\\
s[p(k : a \xrightarrow{u} v)] &\leq_p c[p(k : a \xrightarrow{u} v)] && (rule\ 1)\\
s[p(k : a \xrightarrow{u} v)] &<_p c[p(k : a \xrightarrow{c} t)] && (rule\ 2)\\
s[p(k : a \xrightarrow{u} v)] &\leq_p s[p(k : a \xrightarrow{c} t)] && (rule\ 3)
\end{aligned}
$$

This relationship is used to determine which model is the intended model, as formalized in the next definition. The goal of the following preference relation is to minimize higher priority atoms as much as possible.

Definition 3.7 (Intended T-model). Suppose that M and N are two distinct T-models of a database $\langle \Delta, \bar{u} \rangle$. Then N is preferable to M ($N \ll M$) if, for every atom A in $N - M$, there is an atom B in $M - N$ such that $A <_p B$. We write $N \leq M$ if $N \ll M$ or $N = M$. We say that model N is an intended T-model of $\langle \Delta, \bar{u} \rangle$ if there are no T-models of $\langle \Delta, \bar{u} \rangle$ preferable to N. □

Example 3.9. Consider the models in example 3.6. In this case, definition 3.7 suggests that $M_3 \ll M_3'$ *and* $M_3 \ll M_3''$.

Example 3.10. Consider the database $\langle D_2, s\rangle$ *in example 3.4 and the corresponding priority relationship described in example 3.8. For this database, the following candidate models can be identified.*

$$M_2 = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c),\ level(s), \\ order(u,c),\ order(c,s) \end{array} \right\}}_{M_2(s_\perp)}, \underbrace{\{u[p(k:a \xrightarrow{u} v)]\}}_{M_2(u)}, \underbrace{\emptyset}_{M_2(c)}, \underbrace{\{s[p(k:a \xrightarrow{u} v)]\}}_{M_2(s)} \rangle$$

$$M_2' = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c),\ level(s), \\ order(u,c),\ order(c,s) \end{array} \right\}}_{M_2'(s_\perp)}, \underbrace{\{u[p(k:a \xrightarrow{u} v)]\}}_{M_2'(u)}, \underbrace{\{c[p(k:a \xrightarrow{c} t)]\}}_{M_2'(c)}, \underbrace{\emptyset}_{M_2'(s)} \rangle$$

$$M_2'' = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c),\ level(s), \\ order(u,c),\ order(c,s), p(j) \end{array} \right\}}_{M_2''(s_\perp)}, \underbrace{\{u[p(k:a \xrightarrow{u} v)]\}}_{M_2''(u)}, \underbrace{\{c[p(k:a \xrightarrow{c} t)]\}}_{M_2''(c)}, \underbrace{\emptyset}_{M_2''(s)} \rangle$$

The question is which is the intended model. There is only one atom in $M_2 - M_2'$, $A = s[p(k:a \xrightarrow{u} v)]$, *and there is an atom,* $B = c[p(k:a \xrightarrow{c} t]$ *in* $M_2' - M_2$, *such that* $A <_p B$ *so* M_2 *is preferable to* M_2'. *In fact the same applies for* M_2 *and* M_2'', *so* M_2 *is preferable to* M_2''. □

Lemma 3.1 (Intendedness of Some T-Models). Let $\langle \Delta, \bar{u}\rangle$ be a level $\bar{u}$ database in which there are no cautious b-atoms. Then $\langle \Delta, \bar{u}\rangle$ has a unique minimal T-model, which is an intended T-model. □

The ground clauses of a consistent database can be partitioned into levels based on the following relationship.

Definition 3.8 (Partitioned Database). Let $\langle \Delta, \bar{u}\rangle$ be a consistent level $\bar{u}$ database. For all clauses $A \leftarrow B$ and $A' \leftarrow B'$, if $\mu(A) < \mu(A')$ then $(A \leftarrow B) <_l (A' \leftarrow B')$.

Let $D_1, \ldots, D_k$ be a partitioning of the ground clauses of a consistent database $\langle \Delta, \bar{u}\rangle$ that satisfies the $<_l$ relationship described above. Then we say that $\langle \Delta, \bar{u}\rangle$ has k partitions. □

Example 3.11. Consider the database in example 3.4, the ground clause $s[p(k : a \xrightarrow{c} t)] \leftarrow c[p(k : a \xrightarrow{c} t)] \ll cau$ *is in a lower partition than* $s[p(k : a \xrightarrow{u} v)] \leftarrow c[p(k : a \xrightarrow{u} v)] \ll cau$ *because* $s[p(k : a \xrightarrow{c} t)]$ *is more specific than* $s[p(k : a \xrightarrow{u} v)]$.

Example 3.12. Again, consider the level s database $\langle D_2, s\rangle$ *and its mapping in example 3.4. This database can be partitioned as follows.*

$$P_1 = \left\{ \begin{array}{l} level(u),\ level(c),\ level(s),\ order(u,c), order(c,s),\ c[p(k:a \xrightarrow{u} v)], \\ c[p(k:a \xrightarrow{c} t)] \leftarrow p(j), s[p(k:a \xrightarrow{c} t)] \leftarrow c[p(k:a \xrightarrow{c} t)] \ll cau \end{array} \right\}$$

$$P_2 = \left\{ s[p(k:a \xrightarrow{u} v)] \leftarrow c[p(k:a \xrightarrow{u} v)] \ll cau \right\}$$

Consequently, the level s database $\langle D_2, s\rangle$ *has 2 partitions.* □

Lemma 3.2 (Unique Minimal Extension). Let $\langle \Delta, \bar{u}\rangle$ be a consistent level $\bar{u}$ database and $D_1, \ldots D_{k+1}$ be a partitioning of the ground clauses of $\langle \Delta, \bar{u}\rangle$. Suppose $D_1, \ldots D_k$ has a unique intended model M_k. Then there exists a unique minimal extension of M_k to a unique intended model M_{k+1} of $\langle \Delta, \bar{u}\rangle$. □

Theorem 3.1 (Intendedness of T-models). Let $\langle \Delta, \bar{u}\rangle$ be a consistent level $\bar{u}$ database. Then $\langle \Delta, \bar{u}\rangle$ has exactly one intended T-model, which is denoted M_Δ. For every T-model M, we have $M_\Delta \ll M$. □

Theorem 3.2 (Independent of Partitioning). Let $\langle \Delta, \bar{u}\rangle$ be a consistent level $\bar{u}$ database. Then, M_Δ does not depend on the particular partitioning chosen for $\langle \Delta, \bar{u}\rangle$. □

Consequently, every consistent MultiLog database has an *intended* T-model, and has at least one T-model, namely M_Δ. The declarative semantics of a MultiLog database is its intended T-model.

3.3 Fixpoint Theory

In this section, we present a constructive way of defining the least T-model for a MultiLog level $\bar{u}$ database $\langle \Delta, \bar{u}\rangle$. The key idea is to construct the least T-model $\mathbf{M}_\Delta$ of a database $\langle \Delta, \bar{u}\rangle$ by means of a bottom-up least fixpoint computation based on an immediate consequence operator $\mathbf{T}^{\bar{u}}_\Delta$. Since our T-interpretations are tuples of interpretations, we define $\mathbf{T}^{\bar{u}}_\Delta$ in terms of the immediate consequence transformation of each of the levels in $\langle \Delta, \bar{u}\rangle$.

Definition 3.9 (Fixpoint Operator). Let Δ be a closed consistent database at level $\bar{u}$ and let $\widehat{\Delta} = \langle \widehat{\Lambda}, \widehat{\Sigma}, \widehat{\Pi}, \widehat{\mathcal{Q}}\rangle$ be its Herbrand instantiation defined as usual. Let I be an Herbrand interpretation for Δ. We define $\mathbf{T}^{\bar{u}}_\Delta$ to be the immediate consequence operator such that $\mathbf{T}^{\bar{u}}_\Delta = \langle T^{\bar{u}}_\Delta(I(s)) : s \in \mathcal{S}\rangle$. The operator $T^{\bar{u}}_\Delta$ for each component $I(s) \in I$ is defined similar to the classical case as $T^{\bar{u}}_\Delta : \mathcal{P}(\mathcal{H}) \mapsto \mathcal{P}(\mathcal{H})$, such that

$$T^{\bar{u}}_\Delta(I(s)) = \{A \mid A \leftarrow G \in \widehat{\Delta}, depth(A) = s \text{ and } I(s) \models_{I,\bar{u}} G\}$$

Example 3.13. Consider the level s database $\langle D_4, s\rangle$ *derived from database* D_3 *in example 3.6 by adding the rule* $r_{13} : q(j)$ *in* Π_{D_3}. *For the database* $\langle D_4, s\rangle$, *the intended T-model* M_4 *may be identified as follows:*

$$M_4 = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c),\ level(s),\ q(j), \\ r(j),\ p(j),\ order(u,c),\ order(c,s) \end{array} \right\}}_{M_4(s_\perp)}, \underbrace{\{u[p(k : a \xrightarrow{u} v)]\}}_{M_4(u)}, \underbrace{\{c[p(k : a \xrightarrow{c} t)]\}}_{M_4(c)}, \underbrace{\emptyset}_{M_4(s)} \rangle$$

However, if we consider the sets computed at every stage of $T^{\bar{u}}_{\Delta}$, *we have the following sequence,*

$\delta_1 = \{level(u),\ level(c),\ order(u,c), q(j),\ u[p(k : a \xrightarrow{u} v)]\}$
$\delta_2 = \{r(j),\ s[p(k : a \xrightarrow{u} v)]\}$
$\delta_3 = \{p(j)\}$
$\delta_4 = \{c[p(k : a \xrightarrow{c} t)]\}$
giving us the T-model

$$M'_4 = \langle \underbrace{\left\{ \begin{array}{l} level(u),\ level(c), \\ level(s),\ q(j), r(j),\ p(j), \\ order(u,c), order(c,s) \end{array} \right\}}_{M'_4(s_\perp)}, \underbrace{\{u[p(k : a \xrightarrow{u} v)]\}}_{M'_4(u)}, \underbrace{\{c[p(k : a \xrightarrow{c} t)]\}}_{M'_4(c)}, \underbrace{\{s[p(k : a \xrightarrow{u} v)]\}}_{M'_4(s)} \rangle$$

which is not intended as the component model $M'_4(s)$ *is not minimal, i.e.,* $M'_4(s) \not= \emptyset$. *As such the query returns the answer* true. □

¿From the example, it is obvious that $\mathbf{T}^{\bar{u}}_{\Delta}$ is not monotonic. If we can prove that $\mathbf{T}^{\bar{u}}_{\Delta}$ is monotonic in restricted circumstances, then we can take the fixpoint of $\mathbf{T}^{\bar{u}}_{\Delta}$ in those circumstances.

Theorem 3.3 (Monotonicity of $\mathbf{T}^{\bar{u}}_{\Delta}$). Let $\langle \Delta, \bar{u}\rangle$ be a consistent level $\bar{u}$ database, and $D_1, \ldots, D_k$ be a partitioning of the ground clauses as described in section 3.2

1. $\mathbf{T}^{\bar{u}}_{D_1}$ is monotonic.
2. Let M_{k-1} be the T-model of $D_1 \cup \ldots \cup D_{k-1}$. $\mathbf{T}^{\bar{u}}_{D_k}$ is monotonic if I is initially M_{k-1}.

The monotonicity of $\mathbf{T}^{\bar{u}}_{\Delta}$ guarantees the existence of a fixpoint. As such, the bottom-up fixpoint computation is defined as follows:

$$\begin{aligned} \mathbf{T}^{\bar{u}}_{\Delta} \uparrow^0 (I) &= I \\ \mathbf{T}^{\bar{u}}_{\Delta} \uparrow^{n+1} (I) &= \mathbf{T}^{\bar{u}}_{\Delta} (\mathbf{T}^{\bar{u}}_{\Delta} \uparrow^n (I)) \\ \mathbf{T}^{\bar{u}}_{\Delta} \uparrow^{\omega} (I) &= \sqcup_{n<\omega} \mathbf{T}^{\bar{u}}_{\Delta} \uparrow^n (I). \end{aligned}$$

Let $D_1, \ldots, D_n$ be a partitioning on the clauses of database $\langle \Delta, \bar{u}\rangle$. Then a T-interpretation, M^*_{Δ} is computed using the fixpoint operator to find a fixpoint of $\mathbf{T}^{\bar{u}}_{D_1}$, $\mathbf{T}^{\bar{u}}_{D_2}$ and so on.

$M_1 = \mathbf{T}^{\bar{u}}_{D_1} \uparrow^{\omega} (\emptyset)$
$M_i = \mathbf{T}^{\bar{u}}_{D_i} \uparrow^{\omega} (M_{i-1})$, for $1 < i \leq n$
$M^*_{\Delta} = M_n$.

Example 3.14. Consider the database in example 3.8. The ground rule $s[p(k : a \xrightarrow{u} v)] \leftarrow c[p(k : a \xrightarrow{u} v)] \ll cau$ *is in a higher partition so the fixpoints of the other partitions are found first, and the body of this rule is no longer satisfied, so neither is the head.*

Example 3.15. Consider the level s database $\langle D_2, s\rangle$ *in example 3.4 and its partitioning as presented in example 3.12. Using the fixpoint operator, we proceed as follows in order to compute its T-model* $M^*_{D_2}$.

$$
\begin{array}{ll}
\mathbf{T}^s_{D_2} \uparrow^0 (\emptyset) & = \emptyset \\
\mathbf{T}^s_{D_2} \uparrow^1 (\emptyset) & = \{level(u), level(c), level(s), order(u,c), order(c,s), c[p(k : a \xrightarrow{u} v)]\} \\
\mathbf{T}^s_{D_2} \uparrow^\omega (\emptyset) & = \mathbf{T}^s_{D_2} \uparrow^1 (\emptyset) \\
M_1 & = \mathbf{T}^s_{D_2} \uparrow^\omega (\emptyset)
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{T}^s_{D_2} \uparrow^0 (M_1) & = M_1 \\
\mathbf{T}^s_{D_2} \uparrow^1 (M_1) & = \{s[p(k : a \xrightarrow{u} v)]\} \\
\mathbf{T}^s_{D_2} \uparrow^\omega (M_1) & = M_1 \cup \mathbf{T}^s_{D_2} \uparrow^1 (M_1) \\
M^*_{D_2} & = \mathbf{T}^s_{D_2} \uparrow^\omega (M_1).
\end{array}
$$

Each M_i is a fixpoint of $\mathbf{T}^s_{D_i}$, and we show below that $M^*_{D_2}$ is a rather special T-model of $\langle D_2, s\rangle$.

Theorem 3.4 (Soundness and Completeness). Let $\langle \Delta, \bar{u}\rangle$ be a consistent level $\bar{u}$ database. Then M^*_Δ does not depend on the particular partitioning chosen for $\langle \Delta, \bar{u}\rangle$, and M^*_Δ is the intended T-model of $\langle \Delta, \bar{u}\rangle$. □

We have the following theorem as a corollary which shows the equivalence between the model theoretic semantics and the proof theory [8].

Theorem 3.5 (Equivalence). Let $\langle \Delta, \bar{u}\rangle$ be a level $\bar{u}$ database, $\mathbf{M}_\Delta$ be its intended T-model, G be any ground goal, $\vdash$ be the provability relationship introduced in [8], and ϵ be the empty substitution. Then, we have

$$\langle \Delta, \bar{u}\rangle \vdash_\epsilon G \iff \mathbf{M}_\Delta \models_{\bar{u}} G$$

4 Conclusion

To our knowledge, MultiLog is the first logic based query language for MLS databases. It provides support for multiple belief models and ad hoc belief reasoning. It is free from security breaches such as surprise stories identified in [8]. It supports the possibility of tailoring belief functions according to application needs making it incremental and customizable.

On the theoretical front, we have developed a sound and complete proof procedure and fixpoint operator to constructively compute intended models of databases. We have utilized a critical connection between the concept of inheritance in object-oriented languages and the views at different levels in a MLS

database. This connection has helped us develop the logical semantics presented in this paper. The establishment of the equivalence of the three logical characterizations – proof theoretic, model theoretic and fixpoint – in the context of MLS deductive databases is significant from a theoretical perspective, as it gives insight into the understanding of the logical behavior and mathematical foundations of the language. In the context of databases, the model theoretic semantics and its fixpoint procedure as we proposed here is significant as it now enables us to compute the semantics of MLS databases in a bottom-up fashion.

References

1. P. Bonatti, S. Kraus, and V. S. Subrahmanian. Foundations of secure deductive databases. *IEEE TKDE*, 7(3):406–422, 1995. 131
2. F. Cuppens. Querying a multilevel database: A logical analysis. In *Proc. of the VLDB Conference*, pages 484–494, 1996. 130, 131
3. D. E. Denning, T. F. Lunt, R. R. Schell, M. Heckman, and W. R. Shockley. A multilevel relational data model. In *IEEE Symp. on Security and Privacy*, 1987. 130
4. G. Dobbie and R. Topor. A Model for Sets and Multiple Inheritance in Deductive Object-Oriented Systems. In *Proc. DOOD*, pages 473–488, December 1993. 131
5. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symp. on Security and Privacy*, pages 31–42, 1997. 131
6. S. Jajodia and R. Sandhu. Toward a multilevel secure relational data model. In *ACM SIGMOD*, pages 50–59, 1991. 130, 131, 134, 138
7. H. M. Jamil. Implementing abstract objects with inheritance in Datalog^{neg}. In *Proc. of the VLDB Conf.*, pages 56–65, 1997. 131
8. H. M. Jamil. Belief reasoning in MLS deductive databases. In *ACM SIGMOD*, pages 109–120, 1999. 130, 131, 133, 137, 138, 146
9. H. M. Jamil and G. Dobbie. On the logical foundations of multi-level secure databases. Technical Report TR-IDB-2002-03, Department of Computer Science, Mississippi State University, USA, May 2002. In preparation. 131
10. N. A. Jukic and S. V. Vrbsky. Asserting beliefs in MLS relational models. In *Sigmod Record*, pages 30–35, Ithaca, NY, 1997. ACM Press. 131
11. V. Kessler and G. Wedel. AUTLOG – an advanced logic of authentication. Manuscript. 131
12. M. Kifer, G. Lausen, and J. Wu. Logical Foundations for Object-Oriented and Frame-Based Languages. *JACM*, 42(3):741–843, July 1995. 131, 133
13. T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The seaview security model. *IEEE Trans. on Soft. Engg.*, 16(6):593–607, 1990. 130
14. D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(1/2):79–108, jan/mar 1989. 136
15. A Spalka. Fundamental forms of confidentiality in deductive databases. Manuscript. 131
16. W. Winiwarter. Why is deduction required for database systems ? - some case studies. In *Proc. of the 2nd Data Engineering Forum*, Tokyo, Japan, November 1995. 131

Propagation Completeness of Reactive Constraints

Michael J. Maher[1,2]

[1] School of Computing and Information Technology
Griffith University, Brisbane, Australia
[2] Department of Mathematical and Computing Sciences
Loyola University Chicago
mjm@cs.luc.edu

Abstract. We develop a framework for addressing correctness and timeliness-of-propagation issues for reactive constraints – global constraints or user-defined constraints that are implemented through constraint propagation. The notion of propagation completeness is introduced to capture timeliness of constraint propagation. A generalized form of arc-consistency is formulated which unifies many local consistency conditions in the literature. We show that propagation complete implementations of reactive constraints achieve this arc-consistency when propagation quiesces. Finally, we use the framework to state and prove an impossibility result: that CHR cannot implement a common relation with a desirable degree of timely constraint propagation.

1 Introduction

A key design point of constraint programming languages [17,31,24] is the clean interface between the constraint solver function and the other programming language features, such as control flow, data structures, etc. This enables the semantics of such languages to be defined in two independent parts; essentially the constraint solver is a parameter of the language [16,18].

Nevertheless, there has been a need for both system developers and users to define and implement new constraints. Generally these new constraints are built using the constraint solver interface, while still appearing to be constraints to the programming language. These are the global constraints of CHIP [3], the demons of Eclipse and CHIP [12], the hard constraints of CLP($\Re$) [21], as well as user-defined constraints definable through the use of indexicals in clp(FD) [10], Constraint Handling Rules (CHR) [13], and the general programming features of AKL [9] and Oz [32].

Constraints, in constraint programming languages, are relations that have a built-in behavior provided by a constraint solver. Since we will be discussing the implementation of new constraints in terms of the constraints in the underlying constraint domain, there is likely to be some ambiguity about the word "constraint". For this reason, we refer to the constraint that we wish to implement as a property.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 148–163, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Definition 1. *A* property *is a relation over the set of values $\mathcal{D}$ of the constraint domain.*

A property may be implemented in different ways and in different languages. We propose to abstract from these details and characterize the implementation purely in terms of its effect on the constraint store. We will assume that implementations are simply transducers (or propagators) of constraints, that may react to a strengthening of the constraint store by adding further constraints. This preserves the important property of monotonicity of the constraint solver [18]. The resulting behavior is called constraint propagation. We call the implementation a reactive constraint.

Definition 2. *A* reactive constraint *is an object that reacts to an environment of constraints by adding constraints to the environment.*

All implementations of global and user-defined constraints of which I am aware satisfy this definition, but it is conceivable that an implementation might not. In general, we might want to consider non-determinate reactive constraints. However, for the purposes of this paper it is sufficient to consider determinate reactive constraints. Consequently it is sufficient to represent a reactive constraint by a function, mapping an initial constraint store to an augmented constraint store.

This paper proposes a framework in which to analyze issues of correctness and timely propagation of constraints for reactive constraints, whether global constraints or user-defined. The framework follows the philosophy of the CLP scheme [16] in making the constraint domain a parameter of the framework. Thus this work applies more widely than simply to finite domains.

The framework also adapts some ideas from the theory of concurrent constraint programming (CCP). In particular, the "resting point" or quiescence semantics for determinate CCP programs [29] is the basis of the abstract treatment of reactive constraints.

A key element of the framework is a declarative specification of timely propagation of constraints by a reactive constraint. This is refined into more specialized versions of the same concept. They provide a way to document the degree of timely propagation achieved by an implementation, without revealing the implementation itself. Hence we have the basis for comparing different implementations of the same property by providing standards for the amount of propagation achieved.

Furthermore, these standards document the degree to which a programmer can reason declaratively about the execution of reactive constraints. For example, in a goal $\ldots, y = x + c, z = |x - y|, \ldots$ where $z = |x - y|$ is implemented by a reactive constraint, we would like to be able to conclude that execution will produce $z = |c|$ based only on the declarative meaning of the constraints. But this is only possible if certain performance guarantees are provided by the reactive constraint. (A suitable standard in this case is Eqn-propagation completeness; see Section 3.)

In the next section we briefly introduce constraints, CHR, and elements of the theory of CCP, to provide the technical background for the remainder of the paper.

Section 3 formulates soundness and weak completeness of reactive constraints in implementing properties. It then defines and refines the notion of propagation completeness, which specifies timely propagation of constraints. Here timely refers to the question of whether a reactive constraint reacts to a new state of the constraint store when, in theory, a reaction is possible, or whether it delays reaction until the constraint store is stronger. This is a key concept in the remainder of the paper.

It is natural to expect a close relationship between local consistency conditions and constraint propagation. In Section 4 we formulate a generalized form of arc-consistency, appropriate to the framework, that is parameterized by the underlying constraint domain and a subclass of constraints. It unifies a wide range of consistency conditions in the literature, including interval consistency used in FD solvers [31], hull consistency used for nonlinear real constraints [6], and rule consistency [2]. These consistency conditions are *variable-oriented* in that they are based on the set of values a variable might take. The generalized arc-consistency broadens the notion of consistency to a variety of new consistency conditions. We show that timely constraint propagation achieves the generalized arc-consistency when propagation quiesces.

Concurrent languages have a history of use in implementing reactive constraints. Naish [25] used a coroutining logic language and Saraswat [26] a concurrent logic language to implement value propagation. The first concurrent constraint language [23] was motivated by the idea of programming only with reactive constraints (although not phrased in those terms). Later work on AKL and Oz developed a more complete programming paradigm, and Constraint Handling Rules (CHR) [13] were designed specifically for user definition of reactive constraints. Such languages appear natural choices for defining reactive constraints

In Section 5 we perform a case study of the CHR implementation of a simple, but useful, property: the 3-argument `minimum` relation. Surprisingly, we show that no CHR implementation of this relation is able to achieve a desirable level of timely propagation of constraints on important constraint domains. This demonstrates a limitation of CHR, although it appears that there might be an inherent difficulty in achieving timely propagation of constraints for `minimum` in any language in the CCP tradition.

2 Constraints and Concurrent Constraint Programming

The class of constraints to be used in a constraint programming language and their intended meaning is defined by a *constraint domain*: a pair $(\mathcal{D}, \mathcal{L})$ [17]. Let Σ be the signature – the set of symbols whose meanings are predefined in the language, which we assume includes the equality symbol $=$. $\mathcal{D}$ is a Σ-structure where $=$ is interpreted as identity, and $\mathcal{L}$ is a class of first-order Σ-formulas

closed under variable renaming, conjunction and existential quantification. These formulas are called *constraints*. Constraints that are not composed from other constraints by conjunction, quantification, etc are called *primitive constraints*. We assume that $\mathcal{L}$ contains a constraint $false$ that is never true. In general, Σ is many-sorted but for simplicity we will consider only constraint domains with a single sort. For this paper, we will focus on constraints occurring as part of a conjunction of constraints. We will consider only complete constraint solvers.

An alternative presentation of the meaning of constraints is via a constraint system [29], which is widely used in the theory of concurrent constraint programming. Such a formalization, however, is too abstract for our purposes. It obscures the declarative meaning of the constraints. Since this meaning is at the center of the approach to reactive constraints, constraint systems are not suitable.[1] In particular, it is unable to express interaction between properties and constraints, nor express that a variable is determined by a constraint.

A constraint is *trivial* if it is equivalent to *true*. $\exists_{-\tilde{x}}\phi$ denotes the existential quantification of all variables in ϕ except for $\tilde{x}$. $\tilde{\exists}\ \phi$ denotes the existential quantification of all variables in ϕ.

A constraint c *determines* a variable x which occurs free in c if there is exactly one value of x for which c is satisfiable. That is, if $\mathcal{D} \models \tilde{\exists}\ c$ and $\mathcal{D} \models c(x_1) \wedge c(x_2) \rightarrow x_1 = x_2$. In the case where c determines x, we will refer to the value of x by $\hat{x}$.

An atom has the form $p(t_1, \ldots, t_n)$ where t_i are terms and $p \not\in \Sigma$.

The languages we will investigate are the concurrent constraint programming (CCP) languages [23,27,8]. Constraint Handling Rules (CHR) [13] extends the usual CCP languages by allowing two (or more) atoms in the head of rules. CHR rules have the form[2]

$$\texttt{H}_1\texttt{, ..., H}_k\ \texttt{<=>}\ \ \texttt{c |}\ \ \texttt{B}$$

where $\texttt{H}_1\texttt{, ..., H}_k$ are atoms (the head of the rule), `c` is a constraint, and `B` is a conjunction of atoms and constraints. A program executes by rewriting a *goal* $\langle S, C\rangle$ (a conjunction of atoms S and a constraint store C) according to rules in the program. A rule can "fire" and rewrite a goal $\langle S, C\rangle$ if S contains atoms $H'_1, \ldots, H'_k$ such that $\mathcal{D} \models C \rightarrow \exists \tilde{y} \bigwedge_i (H_i = H'_i) \wedge c$; in that case the atoms $H'_1, \ldots, H'_k$ are replaced in the goal by B. CCP rules can be considered a special case of CHR rules where there is only one atom in the rule head. CHR, which was designed for defining reactive constraints, is intended to extend conventional constraint logic programming systems like Eclipse and Sicstus Prolog with constraints beyond those provided by the underlying constraint solver.

To simplify the exposition, and in line with our assumption of determinate reactive constraints, we address only confluent CCP and CHR programs [1]. These

[1] Nevertheless, it might be possible to add extra structure to constraint systems to reflect the needed properties. Ultimately this might produce a more general theory, but at the expense of a more difficult exposition.

[2] In fact, there are other kinds of rules in CHR, but their behavior can be simulated with this kind.

are programs that always quiesce in the same state (modulo irrelevant internal details). While formally, in a concurrency setting, these classes of programs have restricted expressiveness, it is not at all clear that there is any restriction in ability to express reactive constraints satisfying completeness properties.

As a result of this simplification, we can use the elegant closure semantics of [29] for CCP programs and its extension to CHR programs. Thus the semantics of a conjunction of atoms is a function which maps an initial constraint store to the constraint store that results when no further execution of the program can affect the variables in those atoms. By confluence, this function is well-defined, and we denote it by $\mu[\![\mathcal{R}_1, \ldots, \mathcal{R}_k]\!]$ where $\mathcal{R}_i$ refers to an atom with an associated program. (This semantics is compositional for confluent CCP [28], but not for confluent CHR.) Clearly this semantics is close to the behavior of reactive constraints, which is perhaps why these languages have been used so successfully to define reactive constraints.

In the theory of CCP, a process is quiescent when it reaches a state such that further computation does not impose further constraints on the original variables. Quiescence includes termination, failure, suspension, and active execution that affects only the process's internal state. Once a process reaches quiescence it can only affect the environment after the environment changes. In the following definition of quiescence for reactive constraints this notion is further refined to permit execution to add constraints that may modify the environment if they do not affect constraints in a class $\mathcal{C}$.

Definition 3. *Execution of a goal $\mathcal{R}_1, \ldots, \mathcal{R}_k$ with free variables $\tilde{x}$ in an environment E is* quiescent *wrt a class of constraints $\mathcal{C}$ if, for any further constraints c that the goal imposes, there is no constraint c' in $\mathcal{C}$ such that*

$$\mathcal{D} \models \exists_{-\tilde{x}}(E \wedge c) \rightarrow c' \text{ and } \mathcal{D} \models \exists_{-\tilde{x}} E \not\rightarrow c'$$

Thus, once execution of $\mathcal{R}$ reaches quiescence wrt $\mathcal{C}$, it may contribute further to the constraint store, but will not contribute any constraints that can be viewed as adding a non-trivial constraint from $\mathcal{C}$.

As a consequence of this definition, $\mathcal{R}$ is quiescent in environment E wrt $\mathcal{L}$ exactly when $\mathcal{D} \models \exists_{-\tilde{x}} \mu[\![\mathcal{R}]\!](\exists_{-\tilde{x}} E) \leftrightarrow \exists_{-\tilde{x}} E$.

3 Constraint Propagation

We abstract from an implemented reactive constraint, to consider simply its behavior as a constraint propagator. The *propagation behavior* of a reactive constraint $\mathcal{R}$ is a function $\mathcal{PB}_{\mathcal{R}}$ which maps constraint stores to (possibly infinite) constraint stores such that if $\mathcal{R}$ begins executing in environment C, then when execution of $\mathcal{R}$ quiesces the environment is $\mathcal{PB}_{\mathcal{R}}(C)$. When $\mathcal{R}$ is clear from the context we will omit the subscript.

In the simplest case, the propagation behavior of a primitive constraint c in a CLP system can be defined by $\mathcal{PB}_c(C) = C \wedge c$.

We admit an infinite constraint store as an outcome of a propagation behavior to model iterative approximation methods and other propagation methods that

may not terminate (for example, interval propagation on non-linear functions using infinite precision arithmetic). If C is an infinite conjunction of constraints, we use $\mathcal{D} \models C \rightarrow c$ to denote that for some finite subset C' of C, $\mathcal{D} \models C' \rightarrow c$.

To further clarify definitions, we will examine constraint propagation in the context of two well-known constraint solvers. The CLP($\Re$) constraint solver [20] is complete for equations and inequalities of linear terms over the real numbers, and delays the handling of other arithmetic constraints. The finite domains (FD) solver [31,10] is complete for constraints over the integers that contain only one variable (bounds and bindings), and employs interval reasoning to (incompletely) handle other constraints. Such a solver is used in CHIP, Oz, and Ilog Solver.

Example 1. The propagation behavior of the property $\mathcal{P} : x = y * z$ as implemented in CLP($\Re$) can be defined as: $\mathcal{PB}(C) = C \wedge x = \hat{y} * z$ if y is determined by C; $\mathcal{PB}(C) = C \wedge x = y * \hat{z}$ if z is determined by C; $\mathcal{PB}(C) = C$ otherwise. (Recall that $\hat{y}$ is the value of y determined by C.)

Example 2. The property $x + y \leq z$ is implemented in FD systems with the following propagation behavior: Given a constraint store C, let $x \in [a_1, a_2]$, $y \in [b_1, b_2]$, $z \in [d_1, d_2]$, be the smallest ranges that C implies on x, y and z. Then $\mathcal{PB}(C) = C \wedge x \leq d_2 - b_1 \wedge y \leq d_2 - a_1 \wedge a_1 + b_1 \leq z$. Note that the resulting constraint might be equivalent to C (if the added bounds are not stronger than the ranges in C) or might be equivalent to $false$ (if an added lower bound is greater than the corresponding upper bound, or vice versa).

These examples involve relatively simple reactive constraints. Consequently, the properties they implement are self-evident and the propagation behavior is easy to grasp without formalization. However, the propagation behavior of global constraints is much more complex. We can define it in terms of the operational behavior: $\mathcal{PB}_{\mathcal{R}}(C)$ is the constraint store (restricted to $vars(\mathcal{R})$) resulting after the execution of $\mathcal{R}$ in environment C quiesces.

For confluent CHR and CCP programs used to specify reactive constraints we similarly define the propagation behavior directly by the operational semantics of the program: $\mathcal{PB}_{\mathcal{R}}(C) = \mu[\![\mathcal{R}]\!](C)$. The property defined by these reactive constraints might be specified separately, or we can use the least model of the declarative semantics [13].

Because unsatisfiability of the constraint store is central to constraint programming, we distinguish two finite behaviors of reactive constraints. A reactive constraint $\mathcal{R}$ *finitely fails* in a context C if $\mathcal{D} \models \mathcal{PB}_{\mathcal{R}}(C) \rightarrow false$. A reactive constraint $\mathcal{R}$ *succeeds* in a context C if $\mathcal{D} \not\models \mathcal{PB}_{\mathcal{R}}(C) \rightarrow false$ and $\mathcal{PB}_{\mathcal{R}}(C)$ is finite.

We now consider notions of correctness for reactive constraints.

Definition 4. *Let $\mathcal{R}$ be a reactive constraint, with propagation behavior $\mathcal{PB}_{\mathcal{R}}$, and let $\mathcal{P}$ be a property. $\mathcal{R}$ is a* sound *implementation of $\mathcal{P}$ if, for every constraint store C and constraint $c \in \mathcal{L}$, if $\mathcal{D} \models \mathcal{PB}_{\mathcal{R}}(C) \rightarrow c$ then $\mathcal{D} \models (\mathcal{P} \wedge C) \rightarrow c$.*

In other words, $\mathcal{R}$ is sound if every inference that the constraint solver makes in a context is valid for the intended meaning $\mathcal{P}$. It is straightforward to see that

the implementations of $x = y * z$ in CLP($\Re$) and $x + y \leq z$ in FD languages are sound.

An appropriate notion of completeness is not as easy to find. We will discuss several definitions. A weak form of completeness is to require that if values for all the variables of $\mathcal{P}$ are given then the propagation behavior of $\mathcal{R}$ produces *true* or $false$ in accordance with $\mathcal{P}$. However, we only need consider values that can be expressed within the constraint language.

Definition 5. *Let $\tilde{x}$ be the free variables of $\mathcal{P}$. $\mathcal{R}$ is a* weakly complete *implementation of $\mathcal{P}$ if, for every constraint store C which determines $\tilde{x}$, if $\mathcal{D} \models \tilde{\exists}\ \mathcal{P} \wedge C$ then $\mathcal{R}$ succeeds in the context C; and if $\mathcal{D} \models \neg\tilde{\exists}\ \mathcal{P} \wedge C$ then $\mathcal{R}$ finitely fails in the context C.*

Weak completeness does not address the question of timely propagation of constraints, that is, the propagation of constraints as soon as they are logically implied by the conjunction of the property and the constraint store. It only addresses the correctness of the implementation when used as a test. The following definition addresses timeliness.

Definition 6. *$\mathcal{R}$ is a* propagation complete *implementation of $\mathcal{P}$ if, for every constraint store C and constraint c, if $\mathcal{D} \models (\mathcal{P} \wedge C) \rightarrow c$ then $\mathcal{D} \models \mathcal{P}\mathcal{B}_{\mathcal{R}}(C) \rightarrow c$.*

Since c can be the constraint $false$, these definitions cover the generation of an unsatisfiable constraint store and the inconsistency of $\mathcal{P}$ and C. In [34] a definition of completeness was proposed that is similar to the above definition in the case that c is $false$; thus it only addresses timeliness of detecting unsatisfiability, and not of constraint propagation. We might call this *failure completeness*. Although detecting unsatisfiability is central to the control flow in constraint programming, the propagation of constraints is important in additional ways, for example, to limit the branching factor of the searches that often occur in constraint programming, or to trigger user-defined constraints.

Furthermore, failure completeness does not support compositional reasoning about reactive constraints. Unless $\mathcal{R}_1, c$ or $\mathcal{R}_2, c$ fails, failure completeness of $\mathcal{R}_1$ and $\mathcal{R}_2$ does not give any information about the behavior of $\mathcal{R}_1, \mathcal{R}_2, c$. We would like to be able to infer the propagation behavior of a conjunction of reactive constraints in a constraint store from our knowledge of the propagation behavior of the individual reactive constraints and the constraint store. Propagation completeness does provide for some compositional reasoning[3].

Example 3. Consider the propagation behavior of the property $\mathcal{P} : x = y * z$ as implemented in CLP($\Re$). If we have values for x, y, and z determined by C then

[3] However, propagation completeness is *not* compositional in the sense that from the propagation completeness of $\mathcal{R}_1$ and $\mathcal{R}_2$, and $\mathcal{D} \models (\mathcal{P}_1 \wedge \mathcal{P}_2 \wedge c) \rightarrow c'$ we cannot necessarily conclude that execution of $\mathcal{R}_1, \mathcal{R}_2, c$ will result in a constraint store that implies c'. For example, the FD implementation of $x + y \leq 1$ and $x - y \leq 1$ executed in the constraint store $x \in [-3, 3], y \in [-3, 3]$ will not generate $x \in [-3, 1]$, even though that is a consequence of the properties.

either of the first two clauses of the definition of $\mathcal{PB}$ applies, so that $\mathcal{PB}(C) = C \wedge x = \hat{y} * z$, say (the other case being symmetrical). The CLP($\Re$) solver will simplify the latter constraint to $\hat{x} = \hat{y} * \hat{z}$. Thus the propagation behavior of this constraint in CLP($\Re$) is also weakly complete.

If we consider the constraint store $C = \{y = z\}$, then $\mathcal{D} \models (\mathcal{P} \wedge C) \rightarrow x \geq 0$. However, $\mathcal{PB}_{\mathcal{P}}(C) = C$, and so the implementation of this property in CLP($\Re$) is not propagation complete.

The finite domain constraint solvers provide an interesting demonstration of the effect the language of constraints can have on propagation completeness.

Example 4. Consider the FD implementation of the property $\mathcal{P} : x + y \leq z$. It is not difficult to see that this implementation is weakly complete, since, given values for x, y and z, the added constraints are true for these values iff $\mathcal{P}$ is true for these values. It takes only little further work to recognize that the FD implementation is propagation complete.

On the other hand, suppose the language $\mathcal{L}$ of constraints for FD solvers is extended to contain equations between variables (e.g. $x = y$)[4]. Suppose the constraint store C is $x = y \wedge x \in [1, 9] \wedge y \in [1, 9] \wedge z \in [1, 3]$. Then $\mathcal{P} \wedge C \rightarrow x = 1$, but most FD solvers produce only $x \in [1, 2] \wedge y \in [1, 2] \wedge z \in [2, 3]$, since they only ensure that x and y have the same interval. Thus these solvers are not propagation complete wrt this larger language of constraints.

However, the implementation of [15] implements the equation $x = y$ by applying the corresponding substitution to all properties such as $x + y \leq z$, and is propagation complete wrt the larger language.

The constraint store C does not arise when considering the smaller language of constraints, since $x = y$ is not in $\mathcal{L}$, and so is considered a reactive constraint.

It is not common that a reactive constraint is propagation complete, as the above examples perhaps demonstrate, so we introduce a class of weaker completeness properties that is parameterized by classes of constraints.

Definition 7. *$\mathcal{R}$ is a* propagation complete *implementation of $\mathcal{P}$ wrt a class of constraints $\mathcal{C}$ if, for every constraint store $C \subseteq \mathcal{C}$ and constraint $c \in \mathcal{C}$, if $\mathcal{D} \models (\mathcal{P} \wedge C) \rightarrow c$ then $\mathcal{D} \models \mathcal{PB}_{\mathcal{R}}(C) \rightarrow c$. We also say that $\mathcal{R}$ is $\mathcal{C}$-propagation complete.*

$\mathcal{R}$ is a propagation complete *implementation of $\mathcal{P}$ with respect to the classes of constraints $\mathcal{C}_1, \mathcal{C}_2$ if $\forall C \subseteq \mathcal{C}_1$ $\forall c \in \mathcal{C}_2$ $\mathcal{D} \models \mathcal{P} \wedge C \rightarrow c$ implies $\mathcal{D} \models \mathcal{PB}_{\mathcal{R}}(C) \rightarrow c$. We also say that $\mathcal{R}$ is $\mathcal{C}_1, \mathcal{C}_2$-propagation complete.*

These definitions provide a coarse gauge with which to measure the degree of timeliness of constraint propagation of an implementation. Their compositionality properties are more limited, but they are more realistic goals for practical implementations. In this paper we will consider the following classes of constraints (the classes are identified by their primitive constraints, but we assume that they are closed under conjunction and existential quantification). We write

[4] Notice that FD solvers are also complete for this constraint domain.

X+Y to denote the class of constraints containing X and Y and closed under conjunction and existential quantification.

Bind	equations between a variable and a variable-free term (bindings)
VEqn	bindings, and equations between two variables
Eqn	all equations between terms
Bound	bindings and inequalities between a variable and a variable-free term
VIneqn	VEqn's, Bound's and inequalities between variables
FSet	constraints of the form $x \in S$, where S is a finite set of values
FInt	constraints of the form $x \in S$, where S is a finite interval of integers

The classes of constraints that are of interest depend very much on the constraint domain and the applications for which the reactive constraints will be used. But even among the arithmetic constraints there are many other classes of constraints to consider, for example the UTVPI and TVPI classes in [19].

As discussed in the previous example, most FD implementations of $x + y \leq z$ are FInt-propagation complete, but not (FInt+VEqn)-propagation complete; [15] is an exception.

Note that if all elements of $\mathcal{D}$ can be described by ground terms then Bind-propagation completeness implies weak completeness.

In unpublished work, Wallace [33] employed an abstract notion of approximation of constraint system, which induced a notion of approximate constraint propagation, as a way to control non-terminating propagation. It corresponds, roughly, to a converse of $\mathcal{L}, \mathcal{C}$-propagation completeness, where the only propagated constraints are constraints in $\mathcal{C}$.

Note that even full propagation completeness is not completely satisfactory; it considers the reactive constraint alone, and not in the context of other reactive constraints. Ideally, we would like a much more powerful property, but it appears that such a property can rarely be achieved through constraint propagation.

4 Arc-Consistency

We formulate an extension of the idea of arc-consistency to accommodate an underlying constraint domain, and parameterize it by the notion by a subclass of the constraints. The idea is that a property $\mathcal{P}$ is arc-consistent in a constraint store c if c is strong enough that $\mathcal{P}$ does not add any extra information that can be expressed as a constraint.

Definition 8. *Let $\mathcal{P}(\tilde{x})$ be a property, and let $(\mathcal{D}, \mathcal{L})$ be the constraint domain over which this property is to be implemented. Let c be a constraint and $\mathcal{C} \subseteq \mathcal{L}$ a class of constraints.*

We say that $\mathcal{P}$ is arc-consistent *with environment c wrt $\mathcal{C}$ if for every $c'(\tilde{x}) \in \mathcal{C}$ such that $(\mathcal{P} \wedge c) \rightarrow c'$, we have $c \rightarrow c'$.*

If $\mathcal{C}$ is the entire language of constraints, we simply say $\mathcal{P}$ is arc-consistent with c. A conjunction of properties is arc-consistent with an environment if each property is arc-consistent with that environment.

This definition is implicitly parameterized by the constraint domain. The constraint domain has its affect in the kind of environments that are considered in the definition, since $c \in \mathcal{L}$, as well as the kinds of classes $\mathcal{C}$ permitted. In most cases we will take $\mathcal{C} = \mathcal{L}$, but we introduce $\mathcal{C}$ as a parameter in order to state a later result.

In the conventional setting of CSPs in which arc-consistency was developed, in the constraint domain $(\mathcal{D}, \mathcal{L})$, $\mathcal{D}$ is a set of values and the only constraints in $\mathcal{L}$ are membership by a variable in a finite set of values: the constraints of FSet. In this setting, the above definition reduces to conventional arc-consistency.

If the domain $\mathcal{D}$ is the integers and the constraints are membership by a variable in a finite interval of values (FInt constraints) then the above definition reduces to the interval consistency provided by FD solvers.

Similarly, if we take the constraint domain to be the real numbers with constraints that are inequalities between a variable and a floating point value, then the above definition reduces to the hull-consistency used in solving non-linear real constraints [6,4]. If, instead, the constraints are finite disjunctions of floating point-bounded intervals then the form of consistency used in Echidna [30,5] is obtained. In contrast, box-consistency [5] is not directly a form of arc-consistency, since it is hull-consistency applied to an approximation of $\mathcal{P}$.

Finally, if we only admit bindings $x = a$ as constraints then the above definition gives us rule-consistency [2].

In all the above forms of consistency, it has been understood that an approximation of arc-consistency is being considered. The above formulation shows that these approximations (except for box-consistency) are purely due to the selection of the underlying constraint domain. An abstract formulation of such approximations as approximation spaces [11] can be represented as a language $\mathcal{L}$ of unary constraints, corresponding to the selected subsets of an approximation space.

We may also consider other classes of constraints. For example, if the constraints are linear inequalities over the real numbers then $\mathcal{P}$ is arc-consistent in c exactly when c describes the convex hull of $\mathcal{P}$. Similarly, if the constraints are linear equations over the real numbers then $\mathcal{P}$ is arc-consistent in c exactly when c describes the affine hull of $\mathcal{P}$. Notice that it may not be possible to find a finite arc-consistent environment for a property. For example, if $\mathcal{P}$ is a circle then an arc-consistent environment in the language of linear inequalities must contain an inequality for each tangent of the circle.

One valuable property of this generalized form of arc-consistency is that the primitive constraints involved are not required to be unary. Most forms of consistency, inspired by the original arc-consistency [22], are formulated as restricting the range of values each variable may take - unary constraints. In comparison, the definition above permits the formulation of UTVPI-consistency, for example, where the constraints are unit-two-variable-per-inequality constraints [19].

It is natural to look for a relationship between consistency conditions and constraint propagation. The following theorem demonstrates such a relationship for arc-consistency.

Theorem 1. *Let $\mathcal{R}_1, \ldots, \mathcal{R}_k$ be reactive constraints implementing properties $\mathcal{P}_1, \ldots, \mathcal{P}_k$. Let c be a satisfiable constraint in $\mathcal{C}_1$. Suppose the $\mathcal{R}_i$ are sound and $\mathcal{C}_1, \mathcal{C}_2$-propagation complete.*

If $\mathcal{R}_1, \ldots, \mathcal{R}_k$ is quiescent wrt $\mathcal{C}_2$ in environment c then $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are arc-consistent with c for $\mathcal{C}_2$.

If $\mathcal{P}_1 \wedge \cdots \wedge \mathcal{P}_k$ is arc-consistent with c for $\mathcal{C}_2$, then $\mathcal{R}_1, \ldots, \mathcal{R}_k$ is quiescent wrt $\mathcal{C}_2$ in environment c.

This result exhibits a very close relationship between arc-consistency (as generalized above) and propagation completeness. It says that arc-consistency can be achieved by constraint propagation, provided the implementation is propagation complete. Thus, for example, the implementation of [15] achieves arc-consistency wrt FInt+VEqn.

In many cases we can strengthen the second part of this result. The quiescence of the reactive constraints is *compositional* if $\mathcal{R}_1, \ldots, \mathcal{R}_k$ is quiescent iff each individual $\mathcal{R}_i$ is quiescent. This property holds for CCP programs and perhaps for global constraints, but it is not always true of CHR programs.

Corollary 1. *Under the conditions of the previous theorem, and with the extra assumption that quiescence of the reactive constraints is compositional,*

if $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are arc-consistent with c for $\mathcal{C}_2$ then $\mathcal{R}_1, \ldots, \mathcal{R}_k$ is quiescent wrt $\mathcal{C}_2$ in environment c.

Thus, algorithms for maintaining arc-consistency (in its generalized form) can be viewed as performing propagation complete constraint propagation.

Notice that the arc-consistent state (wrt $\mathcal{C}_2$) that is computed by a $\mathcal{C}_1, \mathcal{C}_2$-propagation complete implementation is not necessarily the weakest (or loosest) arc-consistent state consistent with the original environment.[5] If the implementation can infer constraints outside $\mathcal{C}_2$ then a tighter arc-consistent state will be computed. For example, suppose we take $\mathcal{C}_2$ to be Bind. The property $\mathcal{P} : x^2 = y$ is arc-consistent with the environment *true* wrt $\mathcal{C}_2$. A $\mathcal{C}_1, \mathcal{C}_2$-propagation complete implementation might propagate the constraint $y \geq 0$, which leaves the environment in a more restrictive state (but still arc-consistent wrt $\mathcal{C}_2$). However, if constraint propagation is limited to generating only constraints from $\mathcal{C}_2$, then computation will quiesce in the weakest arc-consistent state.

5 Minimum

Consider the problem of defining a reactive constraint for expressing the minimum of two values, that is, implementing a property $min(X, Y, Z)$ which holds iff Z is the minimum of X and Y, according to an ordering $\leq$. (We assume that

[5] In some treatments, finding the weakest arc-consistent state is implicitly part of the correctness criterion for algorithms maintaining arc-consistency. That is only valid when $\mathcal{C}_2$ contains all constraints, that is, $\mathcal{C}_2 = \mathcal{L}$.

the constraint domain contains a linear ordering expressed by $\leq$.) We can define the property as follows:

$$min(X, Y, Z) \equiv Z \leq X \wedge Z \leq Y \wedge (Z = X \vee Z = Y)$$

We can easily list some sound inferences that can be made with this property:

$$\begin{aligned} min(X, Y, Z) &\Rightarrow Z \leq X, Z \leq Y \\ X \leq U, Y \leq U, min(X, Y, Z) &\Rightarrow Z \leq U \\ U \leq X, U \leq Y, min(X, Y, Z) &\Rightarrow U \leq Z \end{aligned}$$

The first inference is straightforward to implement, and the second is a consequence of the first, assuming the constraint solver incorporates transitivity reasoning (and this is certainly true of a complete solver). The third, however, is more difficult. Nevertheless, it is necessary to achieve appropriate propagation in goals using `min`. For example, if we define

```
min3(W, X, Y, Z) <=>  min(W, X, V), min(V, Y, Z)
```

to compute the minimum of three values, then the goal

```
Y ≤ W, Y ≤ X, min3(W, X, Y, Z)
```

will not return `Z=Y` unless some form of the third inference rule is available.

The following program is a VEqn-propagation complete implementation of the property.

```
min(X, Y, Z) <=>
   Z ≤ X, Z ≤ Y, min2(X, Y, Z).

min2(X, Y, Z) <=> X ≤ Y | Z = X.
min2(X, Y, Z) <=> Y ≤ X | Z = Y.
min2(X, Y, Z) <=> Y = Z | true.
min2(X, Y, Z) <=> X = Z | true.
min2(X, Y, Z) <=> X /=Z | Z = Y.
min2(X, Y, Z) <=> Y /=Z | Z = X.
```

Unfortunately, neither this, nor any other CHR or CCP program, is capable of defining a VIneqn-propagation complete implementation of *min* for CLP($\Re$), as we will now show.

The following result can be proved by induction on the rewrite rules of the operational semantics of CHR and CCP. It shows that the execution of a process with free variables $\tilde{x}$ can generate only constraints on those variables. Consequently, the relation between $\tilde{x}$ and another variable u after the execution of the process is expressed by the conjunction c' of the constraints on $\tilde{x}$ generated by the process and the original constraint store c.

Proposition 1. *Let p be a reactive constraint defined in CHR or CCP.*

If a goal $c, p(\tilde{x})$ can be reduced to an answer constraint c', then $\mathcal{D} \models c' \leftrightarrow (c \wedge \exists_{-\tilde{x}}\ c')$.

To state the following theorem we need a definition. It describes the situation where the "strength" of a constraint lies between that of two other formulas, and the constraint is not equivalent to the weaker formula.

Definition 9. *Given a constraint domain $(\mathcal{D}, \mathcal{L})$, we say a constraint $c \in \mathcal{L}$ is* between *formulas f_1 and f_2 if $\mathcal{D} \models f_1 \rightarrow c$ and $\mathcal{D} \models c \rightarrow f_2$, while $\mathcal{D} \not\models f_2 \rightarrow c$.*

Note that the formulas f_1 and f_2 need not be constraints (i.e. need not be in $\mathcal{L}$).

Now we can state a limitation of CCP and CHR as languages for defining reactive constraints, specifically *min*. For certain constraint domains, it is not possible to implement a particular valid inference concerning *min*.

Theorem 2. *Let $(\mathcal{D}, \mathcal{L})$ be a constraint domain containing $\leq$ such that there is no constraint between $Z \leq X, Z \leq Y, (Z = X \vee Z = Y)$ and $Z \leq X, Z \leq Y$. Then there is no sound implementation of min defined in CCP or CHR over $\mathcal{D}$ that can express the inference*

$$U \leq X, U \leq Y, min(X, Y, Z) \Rightarrow U \leq Z$$

for an arbitrary variable U.

In a constraint domain with purely conjunctive linear arithmetic constraints (i.e. no disjunctive constraints, etc) over the real or rational number, there is no constraint between $Z \leq X, Z \leq Y, (Z = X \vee Z = Y)$ and $Z \leq X, Z \leq Y$. Thus, there is no sound and VIneqn-propagation complete implementation of `min` by CCP or CHR over the constraint domain of CLP($\Re$), for example, and no sound CCP or CHR implementation of `min` that achieves arc-consistency wrt VIneqn. Similarly, simpler constraint domains that provide a total ordering but constraints with fewer or no arithmetic expressions cannot support a VIneqn-propagation complete implementation of `min` in CHR.

On the other hand, constraint domains with non-linear constraints may have a constraint between $Z \leq X, Z \leq Y, (Z = X \vee Z = Y)$ and $Z \leq X, Z \leq Y$. For example, there is the constraint $Z \leq X, Z \leq Y, (Z - X) * (Z - Y) = 0$. Consequently, there is a VIneqn-propagation complete implementation of `min` in CHR over this domain. This comment applies more generally: If the constraint domain is capable of expressing $Z = X \vee Z = Y$ as a single active constraint, then this constraint could be added immediately to the constraint store by an implementation of $min(X, Y, Z)$, thus representing the key inference above.

There are ways of partially overcoming this limitation in CHR, but there is no space in this paper to discuss them.

6 Conclusion

We have introduced a framework for expressing requirements on reactive constraints for timely propagation, and defined a related generalized notion of arc-consistency. Using this framework, we showed the impossibility of implementing a desirable inference rule in a large class of languages.

Statements of the degree of propagation completeness of an implementation provide a way for implementers to make unambiguous claims about timeliness of propagation; this may help users to choose between different implementations. Such statements also provide a clearly defined standard, with which we can perform complexity and other analyses, and hope to prove hardness, optimality and impossibility results (such as the one in this paper).

There are many issues that have been left unresolved concerning this approach to reactive constraints. Among many: Are indeterminate or non-confluent implementations more expressive in terms of timeliness of constraint propagation? It would appear not, but it remains to be proved. Can other forms of consistency, such as path consistency, be generalized and used fruitfully to address the non-compositional elements of CHR? What level of propagation completeness do existing implementations of global constraints achieve?

Acknowledgements

This work has been presented in several workshops and seminars, and has benefited from the comments of many people. I especially want to thank T. Frühwirth, J. Schimpf, M. Wallace, and anonymous referees for their comments. Work on this topic was initiated while the author was employed at the IBM T.J. Watson Research Center. This research was supported by a grant from the Australian Research Council.

References

1. S. Abdennadher, Operational semantics and confluence of constraint propagation rules, *Proc. Int. Conf. on Principles and Practice of Constraint Programming*, LNCS 1330, 252–265, 1997. 151
2. K. R. Apt & E. Monfroy, Constraint Programming viewed as Rule-based Programming, *Theory and Practice of Logic Programming* 1 (6), 713–750, 2001. 150, 157
3. N. Beldiceanu & E. Contejean, Introducing Global Constraints in CHIP, *Mathematical Computer Modelling* 20 (12), 97–123, 1994. 148
4. F. Benhamou, F. Goualard, L. Granvilliers & J.-F. Puget, Revising Hull and Box Consistency, *International Conference on Logic Programming*, 230–244, 1999. 157
5. F. Benhamou, D. A. McAllester & P. Van Hentenryck, CLP(Intervals) Revisited, *International Symposium on Logic Programming*, 124–138, 1994. 157
6. F. Benhamou & W. J. Older, Applying Interval Arithmetic to Real, Integer, and Boolean Constraints. *Journal of Logic Programming* 32(1): 1–24 (1997). 150, 157
7. F. S. de Boer, M. Gabbrielli, E. Marchiori & C. Palamidessi, Proving Concurrent Constraint Programs Correct, *ACM Transactions on Programming Languages and Systems* 19(5): 685–725 (1997).
8. F. S. de Boer & C. Palamidessi, From Concurrent Logic Programming to Concurrent Constraint Programming, in: *Advances in Logic Programming Theory*, Oxford University Press, 1993. 151
9. B. Carlson, M. Carlsson & S. Janson, The Implementation of AKL(FD), *International Symposium on Logic Programming*, 227–241, 1995. 148

10. P. Codognet & D. Diaz, Compiling Constraints in clp(FD), *J. Logic Programming* 27(3): 185–226 (1996). 148, 153
11. A. Colmerauer, Solving the Multiplication Constraint in Several Approximation Spaces, *International Conference on Logic Programming*, 1, 2001. 157
12. M. Dincbas, P. Van Hentenryck, H. Simonis, & A. Aggoun, The Constraint Logic Programming Language CHIP, *Proceedings of the 2nd. International Conference on Fifth Generation Computer Systems*, 249–264, 1988. 148
13. T. W. Frühwirth, Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming* 37(1-3): 95-138 (1998). 148, 150, 151, 153
14. T. Frühwirth, personal communication, 2000.
15. W. Harvey & P. J. Stuckey, Constraint Representation for Propagation, *Proc. Conf. on Principles and Practice of Constraint Programming*, 235–249, 1998. 155, 156, 158
16. J. Jaffar & J.-L. Lassez, Constraint Logic Programming, *Proc. 14th ACM Symposium on Principles of Programming Languages*, 111–119, 1987. 148, 149
17. J. Jaffar & M. J. Maher, Constraint Logic Programming: A Survey, *Journal of Logic Programming 19 & 20*, 503–581, 1994. 148, 150
18. J. Jaffar, M. J. Maher, K. Marriott & P. J. Stuckey, The Semantics of Constraint Logic Programs, *Journal of Logic Programming* 37(1-3): 1–46 (1998) 148, 149
19. J. Jaffar, M. J. Maher, P. J. Stuckey & R. H. C. Yap, Beyond Finite Domains, *Proc. Principles and Practice of Constraint Programming*, LNCS 874, 86–94, 1994. 156, 157
20. J. Jaffar, S. Michaylov, P. Stuckey & R. H. C. Yap, The CLP($\Re$) Language and System, *ACM Transactions on Programming Languages*, 14(3), 339–395, 1992. 153
21. J. Jaffar, S. Michaylov & R. H. C. Yap, A Methodology for Managing Hard Constraints in CLP Systems, *Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 306–316, 1991. 148
22. A. K. Mackworth, Consistency in networks of relations, *Artificial Intelligence*, 8, 99–118, 1977. 157
23. M. J. Maher, Logic Semantics for a Class of Committed-Choice Programs, *Proc. 4th International Conference on Logic Programming,* 858–876, 1987. 150, 151
24. K. Marriott & P. J. Stuckey, *Programming with Constraints : An Introduction*, MIT Press, 1998. 148
25. L. Naish, *Negation and Control in Prolog*, LNCS 238, Springer, 1985. 150
26. V. Saraswat, CP as a General-purpose Constraint-language, *Proc. AAAI-87*, 53–58, 1987. 150
27. V. Saraswat, *Concurrent Constraint Programming,* MIT Press, 1993. 151
28. V. Saraswat & M. Rinard, Concurrent Constraint Programming, *Proc. 17th ACM Symp. on Principles of Programming Languages*, 232–245, 1990. 152
29. V. Saraswat, M. Rinard & P. Panangaden, Semantic Foundation of Concurrent Constraint Programming, *Proc. 18th ACM Symp. on Principles of Programming Languages*, 333–352, 1991. 149, 151, 152
30. G. Sidebottom & W. Havens, Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming, *Comp. Intelligence* 8 (4), 1992. 157
31. P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989. 148, 150, 153
32. P. Van Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, C. Schulte, Logic programming in the context of multiparadigm programming: the Oz experience, *Theory and Practice of Logic Programming*, to appear. 148
33. M. Wallace, Separating Logic and Control in Constraint Programming, 1994. 156

34. J. Würtz, Towards the Verification of Concurrent Constraint Programs in the Field of Finite Domain Reasoning, *Proc. Workshop on Verification and Analysis of Logic Languages*, 87–107, 1994. 154

On Enabling the WAM with Region Support

Henning Makholm[1] and Konstantinos Sagonas[2]

[1] DIKU, University of Copenhagen, Denmark
henning@makholm.net
[2] Computing Science Department, Uppsala University, Sweden
kostis@csd.uu.se

Abstract. Region-based memory management is an attractive alternative to garbage collection. It relies on a compile-time analysis to annotate the program with explicit allocation and deallocation instructions, where lifetimes of memory objects are grouped together in *regions*. This paper investigates how to adapt the runtime part of region-based memory management to the WAM setting. We present additions to the memory architecture and instruction set of the WAM that are necessary to implement regions. We extend an optimized WAM-based Prolog implementation with a region-based memory manager which supports backtracking with instant reclamation, and cuts. The performance of region-based execution is compared with that of the baseline garbage-collected implementation on several benchmark programs. A region-enabled WAM performs competitively and often results in time and/or space improvements.

1 Introduction

High-level languages like Prolog relieve the programmer from worrying about mundane programming details like managing the memory which is needed for a program's execution. Memory allocation happens implicitly by simply creating data structures, and deallocation is the responsibility of the runtime system. The traditional means of doing automatic memory management is *garbage collection* where decisions about what to deallocate are made at run time. Though very sophisticated schemes for collecting garbage efficiently now exist, the process is still potentially time-consuming and hard to predict. It would be desirable to move some of the workload to the compiler. Several proposals for doing *compile-time garbage collection* have been made; see e.g. [8,7] and the references therein.

Region-based memory management [10] takes this principle to the limit. Here all deallocation points in the program are determined by a compile-time analysis, and the runtime system needs only to carry out the preselected actions. Though not all programs are well-suited to having their memory usage reasoned about statically in this way, many are. Moreover, each of the preselected actions operates on a single region and so have a bounded worst-case running time. This makes it easier to guarantee running times of real-time programs.

Region-based memory management was originally proposed for strict functional languages. Much work has been done trying to enlarge the scope of the

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 163–178, 2002.
© Springer-Verlag Berlin Heidelberg 2002

technique to mainstream imperative languages, but so far the only work on adapting it to Prolog has been a preliminary study by the first author [6]. In this paper we take that work a big step further by adding region support to a state-of-the-art WAM-based Prolog implementation and comparing its performance with the same implementation when using a garbage collector. We find that the region-based implementation performs competitively with garbage-collected ones, and in some cases offers significantly better time and/or space behavior.

The next section briefly discusses memory management in the WAM and introduces region-based memory management in general. Section 3 introduces the flavor of our region annotations and region-enabled WAM assembler using a simple example. Section 4 introduces properties of our region model, but we do not describe in detail how to do region inference. Section 5 contains the main contribution of this paper: An abstract machine design for adding region support to the WAM. Section 6 briefly presents the current status of our implementation, and in Sect. 7 we evaluate its performance. Finally, Sect. 8 concludes.

2 Preliminaries and Related Work

2.1 The WAM: Architecture and Heap Memory Management

Due to space limitations, we assume familiarity with the WAM [11]. We depart, albeit only slightly, from the WAM instruction names and adopt the naming convention actually used in our Prolog system: Depending on their classification, variables are denoted as `t` (temporary), `p` (permanent), or `u` (unsafe). Also, instruction names are truncated. So for example, a `putpval` instruction involves a permanent variable and corresponds to WAM's `put_value` instruction.

Besides registers, the WAM memory areas consist of a stack (or stacks) where environments and choice points are maintained, the *global stack* or *heap* where lists, compound terms, and variables that outlive their activation record are stored, and the *trail* that maintains information on variables that need to be reset upon backtracking. Upon backtracking to the topmost choice point, the heap and trail segments allocated after the choice point creation can be *instantly reclaimed*. Perhaps due to this cheap reclamation of memory upon backtracking, the WAM has a reputation of being space-efficient.

However, the instant reclamation provided by the WAM is not a panacea. In reality, Prolog programs are often mostly deterministic and Prolog systems do require additional support for automatic memory management. In most implementations this support comes in the form of heap garbage collection. A lot of work has already been done in this area. An excellent account of issues in Prolog heap garbage collection can be found in [1]; a more recent one appears in [3, Section 3]. As a result, several Prolog systems do have a heap garbage collector—in fact, some of them even have more than one—and it might appear that the issue of heap memory management in the WAM has been solved in a satisfactory way. This impression is often strengthened by the effectiveness of Prolog garbage collectors; garbage collections that recover 90–99% of the heap

space are not unusual. Notice however, that there is another way of interpreting this figure, namely that heap memory *allocation* in the WAM is suboptimal. Regardless of the view that one prefers, the garbage collection process penalizes a program's execution as it happens during run time rather than statically.

2.2 Region-Based Memory Management

Region-based memory management was proposed by Tofte and Talpin [10] as an alternative to garbage collection for functional languages. The basic premise of this technique is that a compile-time analysis called *region inference* annotates the program with explicit instructions for allocation and deallocation of memory. These instructions utilize the *region paradigm*: Memory blocks are grouped together in *regions*. A new block can be allocated in a region at any time, but deallocation can only happen for a region in its entirety. The number of regions varies during the execution of the program and is in principle unbounded. However, the grouping-together of allocations allows a static analysis to keep the number of distinct regions it needs to reason about down to a manageable level.

Several benefits are associated with this scheme:

- During run time, no work is spent on garbage collection (not only collecting the garbage but tracing pointers to find it).
- Because the region inferencer can analyze the possible futures of the computation (whereas a garbage collector typically views the mutator as a black box), it can sometimes deallocate data that GC would consider live.
- The basic region operations can be implemented to all run in constant time—including the deallocation of a region whose size is not statically known. Because there are no GC pauses either, it is possible to reason accurately about the execution time of a region-annotated program in real-time environments.
- Region-based memory management may lead to better cache behavior than garbage collection, because it naturally reuses memory for short-lived objects in a LIFO fashion, whereas garbage collectors usually imply a round-robin usage pattern for the nursery.

There are also certain drawbacks, however. Most prominent is the fact that certain programs are not at all well-suited to static determination of object lifetimes. One such example is an interpreter, whose source code gives no information about the lifetime of the data that represent the interpreted program's data.

Another drawback of the early Tofte–Talpin proposal is that its region inference is not strong enough to handle most real programs with satisfactory results. It is based on the principle that the lifetime of each region must coincide with the evaluation of one source-level expression. In particular, any region that existed at the time of a call must be live though the execution of the entire function-call expression, so the arguments in a tail-recursive call can be deallocated only after the recursion. Several schemes for relaxing this principle have been proposed; the latest one by Henglein, Makholm, and Niss [4], henceforth referred to as the HMN model, is the basis of the region system we employ.

Regions for Prolog. In [6] (and in more detail in [5]), the first attempt to extend region-based memory management to support backtracking and cuts was made. The challenge is that Prolog's control flow makes it difficult to find meaningful places to insert explicit deallocation operations. In a program such as

```
main :- ⟨COMPUTE T SOMEHOW⟩,                                  | foo(1).
        foo(V),  % succeeds twice                             | foo(2).
        ⟨DO SOMETHING WITH T⟩, ❶ ⟨DO SOMETHING WITHOUT T⟩,    |
        bar(V).  % fails the first time we get here           | bar(2).
```

the last use of `T` is when program point ❶ is reached for the *second* time. Ideally, one would like to deallocate (the region of) `T` at that point, but in general the code executing then does not know whether it is running for the first or the second time. It will not do to postpone the deallocation until after the possible failure either, because in less contrived examples than this one it will not be apparent where in the source code the last relevant failure is.

The solution to this is that backtracking should be transparent to regions. Whenever backtracking occurs at run time, it becomes the region management library's job to restore all regions to the state they had when the choice point was created. This includes undoing allocations and region creations made after the choice point (instant reclamation for regions) and recovering regions that the program thought it had deallocated. Algorithms and data structures to do this efficiently in the presence of cuts were described in [6].

The main problem with [6], which this work remedies, is that it is not oriented towards contemporary state-of-the-art implementation models for Prolog. The preliminary performance measurements used an ad hoc Prolog compiler. How to integrate the region operations into a WAM-based Prolog implementation was not addressed. In particular, the handling of conditional bindings inside structures was incompatible with the WAM's data model. In short, although results of [6] show promise, region-based memory management in Prolog *a la* [6] requires a fundamental shift from the abstract machine for Prolog execution: an action which raises concerns (after all, memory management is just a part of a language's implementation) and therefore is a path that most Prolog implementors are probably not willing to take. We address this problem and offer an alternative to [6] which is WAM-based and imposes minimal changes to 'plain' WAM.

3 Compiling with Regions: A Step-by-Step Example

The purpose of this section is three-fold: 1) discuss how issues of region-based memory management translate to the context of WAM-based Prolog, 2) explain our implementation, and 3) introduce our design decisions which are presented in a more detailed manner in Sect. 5.

Consider the familiar naive reverse program shown below:

```
main :- nrev([1,2,3],X), write(X).
nrev([],[]).
nrev([H|T],L) :- nrev(T,V), append(V,[H],L).
append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

Analyzing this program to infer in which regions data should be allocated is a process that requires support from type inference and benefits from information about modes. However, note that the above program contains *no programmer-supplied annotations about modes and types*; it is up to the region inferencer to infer this information.[1] Such a whole program region analyzer could produce the following region- and mode-annotated Prolog program:

```
:- mode main.
main :-
  °new R2, nrev(R2·[1|R2·[2|R2·[3]]],X)°i(R2)°o(R0), write(X), °release R0.
:- mode nrev(i,o).
nrev([],[])°i(R6)°o(R0) :- °release R6, °new R0.
nrev([H|T],L)°i(R6)°o(R1) :-
  nrev(T,V)°i(R6)°o(R4), °new R1, append(V,R1·[H],L)°c(R1,R4), °release R4.
:- mode append(i,i,o).
append([],L,L)°c(R0,R4).
append([H|L1],L2,R0·[H|R0·L3])°c(R0,R4) :- append(L1,L2,L3)°c(R0,R4).
```

In fact, this is exactly the intermediate program produced by our analyzer. The region inferencer has for example inferred that the list `[1,2,3]` will live in a *new* region named R2 which is created before the call to `nrev/2` and is passed to it as an *input* (i) parameter. The result of `nrev/2` will be placed in an *output* (o) region R0 and can be *released* after the call to `write/1`. Finally, there are some regions which are *constant* (c). These are regions that the callee must not release; the caller expects them to be around even after the predicate call returns.

We now perform the following Prolog program transformation: Rather than passing the region parameters as annotations to heads and calls, we pass them as extra arguments. Also, the °new and °release annotations can be considered new Prolog builtins which are treated specially by the compiler. Finally, we also introduce a new compiler builtin called °return that gets produced whenever a °new region annotation would need to create a region that is annotated as output. For example, since the region variable R0 is annotated as output in the first clause of `nrev/2`, rather than creating a call °new(R0), we introduce a new region variable Å0 and translate the call as °new(Å0), °return(Å0,R0). The °return builtin stores the region reference from Å0 into the region variable R0 which has been already unified with the region variable in the caller. We elaborate on need for this in Sect. 5.2. Performing this program transformation results in the following Prolog program:

[1] The only assumption that our analyzer currently makes is that the program contains a zero-arity "top-level" predicate such as `main/0` in our example.

code for main/0	*code for 'nrev/2'/4*		*code for 'append/3'/5*
allocate	switchonlist r3 L_1 L_2	L_2: *% new clause*	switchonlist r3 L_3 L_4
new_trgn r1	try 4 L_1	allocate	try 5 L_3
putlist_trgn r2 r1	trust 4 L_2	getpvar v2 r2	trust 5 L_4
bldnumcon 3		getlist r3	
bldnil	L_1: *% new clause*	unipvar v3	L_3: *% new clause*
putlist_trgn r5 r1	getnil r3	unitvar r3	getnil r3
bldnumcon 2	getnil r4	getpvar v4 r4	gettval r4 r5
bldtval r2	release_trgn r1	putpvar v5 r2	proceed
putpvar v2 r2	new_trgn r1	putpvar v6 r4	
putlist_trgn r3 r1	return_ttrgn r1 r2	call 7 'nrev/2'/4	L_4: *% new clause*
bldnumcon 1	proceed	new_trgn r1	getlist_tvar_tvar r3 r6 r3
bldtval r5		return_tprgn r1 v2	getlist_trgn r5 r1
putpvar v3 r4		putpval v5 r2	uni_tval_tvar r6 r5
call 4 'nrev/2'/4		putpval v6 r3	execute 'append/3'/5
putpval v3 r1		putlist_trgn r4 r1	
call 4 write/1		bldpval v3	
release_prgn v2		bldnil	
dealloc_proceed		putpval v4 r5	
		call 7 'append/3'/5	
		release_prgn v5	
		dealloc_proceed	

Fig. 1. Generated WAM code for the region-annotated naive reverse program

```
:- pragma main.
main :-
  °new(R2),'nrev/2'(R2,R0,R2·[1|R2·[2|R2·[3]]],X), write(X),°release(R0).
:- pragma 'nrev/2'(i,o,w,w).
'nrev/2'(R6,R0,[],[]) :- °release(R6), °new(Å0), °return(Å0,R0).
'nrev/2'(R6,R1,[H|T],L) :- 'nrev/2'(R6,R4,T,V), °new(Å0), °return(Å0,R1),
                           'append/3'(Å0,R4,V,Å0·[H],L), °release(R4).
:- pragma 'append/3'(c,c,w,w,w).
'append/3'(R0,R4,[],L,L).
'append/3'(R0,R4,[H|L1],L2,R0·[H|L3]) :- 'append/3'(R0,R4,L1,L2,L3).
```

Note that the above program now contains very few region annotations. Most of the information on which argument positions correspond to region variables is kept in the form of automatically generated compiler pragmas (*w* denotes a non-region argument position; other letters denote region variables and the pragma information describes their use). This program can be seen as the intermediate code representation that a region-enabled WAM compiler uses. Compiling this program results in the WAM code shown in Fig. 1. Instructions added to the WAM are shown underlined in the figure. Notice the correspondence between region builtins and the new WAM instructions that implement their functionality. Also, note that e.g. the annotation in the second clause of `'append/3'/5`'s last argument has resulted in a getlist_trgn instruction rather than in a getlist.

4 Our Region Model

The region system for Prolog we employ is based on the HMN region model [4] which, in its original formulation, works for first-order functional programs. In the HMN model, the lifetimes of regions are asynchronous with respect to the

call/return discipline of the program. Region handles (the pointers to region control blocks which are used to allocate in, or deallocate, the region) can be passed as parameters in function calls but they cannot be stored in compound terms.

In general, callers pass one or more *input regions* as extra arguments to callees; these are where the ordinary arguments have been441 allocated, and the callee is responsible for deallocating the region after reading the input. Conversely, the callee returns one or more *output regions* to the caller; the ordinary return value is allocated in the output regions, and the caller deallocates them when the return value has been read. Instead of deallocating its input regions, a function (or predicate in the Prolog setting) may return them as output regions, such as if the output value contains parts of the arguments.

In addition to the input and output regions, the HMN model also has a third kind of region parameters, called *constant regions*. These are passed from the caller to the callee, but the callee does *not* deallocate them; rather they can be used for allocating memory for return values. Constant regions are used when the caller needs to be able to specify a preexisting region for allocating a return value (or parts of it). Operationally, a constant region is equivalent to an input region that is always reused by the callee as an output region, but because this reuse *always* happens, the passing-back-as-output can be optimized away in practice. In the region inference process (and in the specialized type system that guarantees the safety of the region-annotated program) constant regions play a special role, but due to space limitations we do not describe those in this paper; the reader is referred to [4] instead.

A final feature of the HMN model that ought to be mentioned here is that regions are *reference counted*. A count of the number of references to each region—not the number of allocations in the region or pointers into it, but the number of pointers to the region control block which can be used to allocate more memory in it or deallocate the region—is kept at run time, and when the last reference goes away, the region is deallocated. There is no explicit deallocation primitive, but rather primitives to increase (*alias*) and decrease (*release*) the reference count.

5 WAM with Region Support

We now describe how to extend a WAM-based Prolog system to support regions.

5.1 Memory Architecture

Our basic premise is that a region-enabled WAM offers regions as an *enhancement* to the WAM heap, not as its *replacement*. This means that the memory architecture contains both a heap and a region area. As shown in Fig. 2, all other memory areas of the WAM are of course still present. Note that, like the heap and the trail, regions are also segmented according to choice points. Shaded areas denote areas which will be reclaimed on backtracking. The figure does not show the data structures used to implement backtracking of regions.

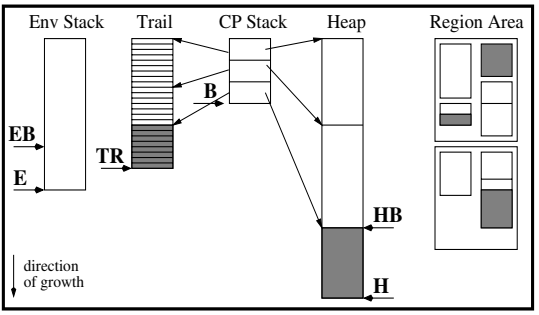


Fig. 2. Memory areas of a region-enabled WAM

In principle, we allow data in the regions to be freely intermixed with data on the heap. A structure on the heap can reference subterms in a region, and vice versa. This design means that existing non-region-annotated code for the libraries, the compiler, and the top-level interactive loop can coexist with region-annotated code in our implementation. In a wider perspective, it also means that, given a sufficiently smart region inferencer, a program could be annotated to allocate short-lived data in regions but still be able to revert to using the garbage-collected heap where regions cannot give acceptably tight lifetimes.

Consequences of Less Structured Memory Layout. On Fig. 2 the region area is shown as being divided into two subareas. That is to indicate that the region area is not necessarily contiguous in memory: New "batches" of memory can be added to it as the need arises, without relocating the existing regions. In fact, each region does not even need to be contiguous; regions are implemented as linked lists of *cards* which might well be from different "batches".

One consequence of this is that the abstract machine cannot enforce a strict spatial relation between the different memory areas, at least not if the implementation is written in (relatively) portable C and does not do its own low-level memory allocation. It is a common optimization trick for WAM implementations to make sure that, say, the local stack is always allocated at higher addresses than the heap, so that a single test can determine whether a pointer points into the stack or the heap. With regions around, there is a third alternative, namely cells pointing into a region area. These pointers should usually be treated as those pointing into the heap, but a single comparison to determine whether they are region or heap pointers does not suffice since region areas can be located on either side of the local stack. Thus, all tests for pointing-into-the-heap in the implementation need to be updated and in a region-enabled WAM become more expensive. Such tests appear in the uni*val instructions and the unification subroutine. They are used to enforce the WAM invariant that there should not be any pointers from the heap to the stack. This invariant extends nicely to regions: there should not be any pointers from the region area to the local stack either.

Instant Reclamation and Conditional Deallocation for Regions. We adopt the principle of [6] that backtracking should be transparent to the region area. This means that recent allocations in the region area (including recent region creations) must be undone upon backtracking, as indicated by shaded

parts of the regions on Fig. 2. Conversely, if a failing computation path releases a region, it must be kept alive and reinstated at backtracking. Techniques for how to do this were developed in [6]; they transfer essentially without change to the WAM-based environment. Unfortunately, space limitations prevent us from presenting the details, and we refer the reader to [6,5]. We only note that the implementation of all *choice* instructions in the WAM must be extended to support backtracking of the region area. It is not sufficient to use specially enhanced choice instructions when compiling a region-annotated program.

Conditional Bindings in Regions. The handling of conditional variable bindings that need to be reset upon backtracking is where we deviate most from [6]. In the WAM, conditional bindings are recorded on a separate *trail* stack, which in its basic form is simply an array of addresses of cells that must be reset. Choice points contain pointers into the trail that determine which part of it is relevant in a given backtracking operation. [6] asserted (wrongly) that it would be very complicated to make a single global trail work well with regions. Instead it proposed a private trail for each region, organized as a linked list of bound variables. This required that two words be allocated for each bindable variable, which conflicts with the WAM's use of interior variables in compound terms.

In our implementation we stick to the global trail; we even intermix trail entries for variables in regions with entries for variables on the WAM stacks. The only care this requires is that we only trail bindings that really are conditional.

A binding is conditional if the variable being bound was allocated *before* the most recent choice point. The WAM mandates that this should be checked for each variable binding, which can be done with two pointer comparisons. Nevertheless, some Prolog systems prefer not to do this checking but to instead record every binding in the trail. In fact, even the trail overflow can be omitted; see [2]. This strategy is acceptable because no variable appears twice on the trail, thus the trail never grows larger than the size of the WAM heap (and stack).

When regions are present, one cannot avoid checking for conditionality anymore. Because region memory can be reused without backtracking taking place, the trail can keep growing without bounds when deterministic code runs, if it also records unconditional bindings.

Checking accurately whether a binding in a region is really conditional is complicated, but an approximation suffices: Instead of checking whether the *variable* is older than the most recent choice point, we check whether the *region card* containing the variable was added to the region before the most recent choice point. This test may give rise to a number of "false positives", but not enough to risk the trail growing unboundedly. The reason for this is that a choice point created after a card was added to the region prevents all of that card from being used more than once until backtracking occurs (or the choice point is cut away, in which case the not-conditional-anymore bindings should be purged from the trail anyway).

To make this check possible, we set aside one word in each card to hold a timestamp, using a "clock" that ticks each time a choice point is created.

Given that cards are properly aligned in memory, the timestamp's address can be recovered from the address of the variable by a simple bit-masking operation.

5.2 Instruction Set

Instructions for Managing Regions. The new_trgn instruction creates a new region (with a reference count of 1) and puts a pointer to its region control block in a specified X register. Similarly, new_prgn creates a region and puts the handle in a specified Y register. All the region-specific instruction come in this kind of pairs; henceforth we will just present instructions as ending with _rgn and thereby understand a pair of _trgn and _prgn instructions.

The alias_rgn instruction increases a region's reference count by one, and the release_rgn decreases the reference count by one, and, if it becomes zero, "deallocates" the region. We put "deallocate" in quotes here because it may be necessary to take special measures to ensure that the region can reappear at backtracking.

The alias_rgn and release_rgn are the only ways for the reference counter to change value, except that when backtracking occurs, the region manager resets all reference counters to their previous values. Thus, the notion that the counter really counts references is just a convention, but serves as a guideline for when to use the instructions.

In practice, the alias_rgn instruction is seldom needed, but the ability to emit it is important as a fall-back strategy of the HMN region inference mechanism. The most common reason for needing it is code such as

```
main :- ⟨COMPUTE T⟩, a(T), a(T).
a(T) :- ⟨USE T⟩, ❷ ⟨DO SOMETHING THAT DOES NOT INVOLVE T⟩.
```

where a/1 can release the region for T after its last use (e.g., at program point ❷). Then main/0 can use an explicit alias to keep T alive during the entire first call to a/1, and pass the (then only) reference to the region to the last call to a/1, such that the data will be deallocated when the last call reaches point ❷.

Instructions for Allocating Data in Regions. Next, we add instructions to allocate data in regions. The principle is that every way to allocate something on the heap should have a corresponding way to instead allocate it in a region. For example, the WAM's puttvar and putuval instructions allocate variables on the heap. Their region-allocating counterparts are puttvar_rgn and putuval_rgn.

Allocation of compound term is not so simple. In the WAM, a typical instruction sequence consists of a getstr or putstr instruction followed by a number of uni* or bld* instructions. Each of the instructions in the sequence allocates a single part of the compound term by increasing the **H** register.

In a region, this principle does not work, because subsequent single-cell allocations in a region will not necessarily be contiguous. Instead, getstr_rgn and putstr_rgn must allocate space for the entire compound term in one operation.

Then the address of the argument cells must be stored in a register for the subsequent uni*/put* instructions to know where to place the arguments. Which register? The most natural choice would be the WAM's **S** register which is already used for a similar purpose in a READ-mode getstr sequence. But in our baseline implementation, the **S** register is already used in WRITE mode—namely, by being set to zero it signals that we are in WRITE mode. So we have to add a new register for the purpose of filling in fresh compound terms. We call it the **W** register.

For simplicity, and to avoid instruction set bloat, we also change the *non*-region-aware getstr and putstr to allocate the entire functor on the heap at once, and put the argument cell address in the **W** register. Then, say, getstr and getstr_rgn sequences can use the same set of uni* instructions, which we change to use the **W** register instead of **H**.

As a further optimization, we use the **S** register instead of **W** in the putstr(_rgn) and bld* instructions, where it is implicit that we are in WRITE mode. This is because on register-poor architectures, such as the x86, it may be possible to register-allocate **S** in the emulator loop, but probably not both **S** and **W**.

When (get,put)str(_rgn) now need to allocate the entire compound term, they must know how big it is. The arity of the functor can be found by looking it up in the symbol table, but we can save that memory reference by giving the number directly in the WAM instruction. Finally, the optimized list instructions (get,put)list are changed and enhanced the same way as the (get,put)str ones.

Instructions for Avoiding Interior Variables. In the WAM, the argument fields of a structure can be free variables. This is a space-saving device for the WAM, but is not always a good thing when regions are around. The reason is sharing. To see the problem, consider this program fragment (which is somewhat idealized, but one can imagine point ❸ in the following code to also contain other predicate calls):

```
main :- blah(P,Q), ❸ ⟨COMPUTE WITH Q ONLY⟩.
blah(foo(V),bar(V)).    % ... perhaps other blah/2 clauses ...
```

The `blah/2` clause creates two structures with the `bar` structure containing a pointer to an interior variable in the `foo` structure. This means that even though `P` is a singleton variable in `main/1`—and is the only reference to the `foo` structure itself—it is *not* safe to deallocate the region containing the `foo` structure before the computation with `Q` is finished (e.g., in point ❸).

This effect means that the region inferencer must take care to keep the `foo` alive as long as the subterm of `bar` may be referenced. In practice, that need interferes with other approximations made by the region inferencer, so it often leads to large losses of space efficiency. Therefore, our design includes another possibility: The above program can be automatically annotated by our region analyzer to allocate `V` in a region different from the one containing the `foo` structure. The `blah/2` clause above can be annotated and compiled as shown below:

```
:- pragma 'blah/2'(c,c,c,w,w).
'blah/2'(R1,R2,R3,R1·foo(R2·V),R3·bar(V)).
```

```
getstr_trgn r4 r1 2 foo/1
unitvar_trgn r1 r2
getstr_trgn r5 r3 2 bar/1
unitval r1
proceed
```

Here the unitvar_trgn instruction allocates a variable in region R2 and fills in a pointer to it in the `foo` structure in R1 that is being added (aside for storing a pointer to the new variable in X register r1, as the unitvar instruction does).

Instructions for Output Regions. As mentioned in Sect. 4, the HMN region model requires that functions can return output regions to their callers alongside their normal return values. A decision needs to be made about how to return output regions in the WAM. At first, this may seem trivial. Prolog natively supports "returning" multiple values from a predicate by simply passing unbound variables in and instantiating them in the called predicate. Why not use this mechanism for output regions, too? One might. But doing it naïvely would be wasteful, because output regions do not need the full generality of Prolog variables. The second clause of the following intermediate code would compile to:

```
:- pragma 'foo/1'(o,w).
'foo/1'(R,a) :-
   °new(R).
'foo/1'(R,T) :-
   'foo/1'(R,V),
   T = R·f(V).
```

```
allocate 2 5
getpvar v2 r1
getpvar v3 r2
putpval v2 r1
putpvar v4 r2
call 5 'foo/1'/2
putpval v3 r1
getstr_prgn r1 v2 2 f/1
unipval v4
dealloc_proceed
```

Here the caller of `'foo/1'/2` passes a *pointer* to a free variable (presumably somewhere on the local stack) in r1. That pointer is stored at location v2 in its own stack frame before calling itself recursively. When the recursive call returns, v2 still contains a pointer to the region reference instead of the region reference itself; therefore getstr_prgn and all other region-annotated allocation instructions must be prepared to *dereference* their region parameters. WAM instructions always dereference their value arguments, but region inputs and outputs are supposed to be strongly moded: We know exactly when the region variable is bound at compile time, so why waste cycles on testing it again at run time?

For performance reasons we choose the following slightly more complicated solution. The caller still passes in an address to a free local stack word for each output region. But in the callee, this address is never used in normal region operations. Instead we introduce two new instructions return_rgn and unreturn_rgn for moving data to and from the pointed-to call, respectively. All region variables other than formal output parameters *always* contain a direct region reference (when they contain anything meaningful at all).

We can now rewrite the above example to use auxiliary variables where the original used a formal output region for a region operation. The intermediate code and the region-enabled WAM code it compiles to are shown below.

```
:- pragma 'foo/1'(o,w).
'foo/1'(R,a) :-
    °new(Å0),
    °return(Å0,R).
'foo/1'(R,T) :-
    'foo/1'(R,V),
    °unreturn(R,Å0),
    T = Å0·f(V).
```

```
    try 2 L5
    trust 2 L6

L5: % new clause
    getcon r2 a
    new_trgn r2
    return_ttrgn r2 r1
    proceed
```

```
L6: % new clause
    allocate 2 5
    getpvar v2 r1
    getpvar v3 r2
    putpval v2 r1
    putpvar v4 r2
    call 5 'foo/1'/2
    unreturn_ptrgn v2 r1
    putdval v3 r2
    getstr_trgn r2 r1 2 f/1
    unipval v4
    dealloc_proceed
```

6 Our Implementation

Our implementation is based on XXX, a WAM-based Prolog system which is a light-weight derivative of the XSB system. XXX is a full Prolog implementation, features a jump-table based bytecode emulator, and comes with both a mark-&-slide and a mark-&-copy heap garbage collector; see [3]. However, it supports tabling exclusively based on CHAT, follows some of the advice on implementing Prolog emulators given in [9] and [2], and its compiler performs instruction merging more aggressively than XSB's. As a result, XXX is a reasonably fast system: On the x86 and on the set of standard Prolog benchmarks, XXX (in the setting *without* region support) is comparable in speed to SICStus 3.8 #4.

A design decision was to do region inference as a source-to-source transformation. Our region inferencer works on a whole program at a time; modular region inference is not currently supported. The original Prolog program is transformed into a region-annotated program in the syntax of the example in Sect. 3. We use region inference algorithms from the original HMN prototype [4][2], adapted to work with Prolog input instead of an SML subset.

The second transformation, which converts region parameters to Prolog-level parameters and introduces auxiliary Å variables for output-region manipulation, is also a separate source-level operation. After that, the annotated program is compiled into bytecodes by an enhanced version of the XXX compiler which recognizes the region annotations. Because of the source-level preprocesses, no drastic changes to the structure of the compiler were necessary. Our runtime system has been extended as described in Sect. 5. In addition to the 45 new instructions for regions, several changes to existing code were necessary:

- A new **W** register has been added.
- Certain pointer comparisons become more complex because the region area may be discontiguous; see Sect. 5.1.
- *Choice* and *cut* instructions must test whether it is necessary to invoke the backtracking/cut code in the region manager.
- Choice points take up four more cells than in XXX, to store administrative data for the region manager.

[2] About half of the 12,000 lines of code in our region inferencer are from the HMN prototype.

Table 1. Time performance comparison (in ms). The rows labeled WAM correspond to the *unmodified* XXX system (i.e., without region support), and include GC times

	queens	dnamatch	rdnamatch	browse	rbrowse	serial	gsort	nreverse
Regions –W	630	8260	8380	6010	5360	1540	1660	1350
Regions	630	9230	9920	5940	5410	1570	1950	1350
WAM –sliding	600	7550	7560	5330	5310	1730	2170	2130
WAM –copying	620	7290	7360	5160	5010	1630	2060	1880
gctime –sliding	0	690	730	690	1290	600	680	750
gctime –copying	0	480	540	490	890	490	550	400

These changes in general make region-less programs run about 5% to 10% slower than on the original XXX emulator (although one benchmark surprisingly runs 5% *faster*).

The region manager code, based on [6], required no major changes apart from integrating it into the XXX runtime system and adapting it to WAM-like trailing. It uses a region card size of 32 words (128 bytes) of which 30 are available for allocation, and allocates cards in batches of 100 cards using `malloc()`.

7 Performance Evaluation

We conducted our experiments on a dual processor Pentium III (Coppermine) 933 MHz machine with 1 GB of RAM and 256 KB of cache running Linux.

To measure the performance of the region-enabled XXX system vs. its plain WAM variant, we used standard Prolog benchmark programs or programs previously used to measure performance of Prolog garbage collectors. In two of them, **browse** and **dnamatch**, we also slightly modified the code to be more region-friendly; the modified programs are referred to **rbrowse** and **rdnamatch** respectively.

Tables 1 and 2 contain results of our experimental evaluation. For the region-enabled system, we used two configurations: one where the region inferencer insists on using the interior variable convention of the WAM (identified with –W), and one where it does not. As mentioned, XXX features two garbage collectors: a non-segment order preserving mark-&-copy (default), and a mark-&-slide collector which traditionally is the one used in the WAM-based Prolog world. In our time comparisons, we use both settings. Times spent in GC are also shown in Table 1.

The **queens** program naturally reclaims heap space by backtracking and needs no GC. Still, the region-enabled system executes it in about the same time as the WAM, while actually requiring less memory space (data shown in Table 2).

A very bad case for regions occurs in the original **dnamatch** program. In this program, the size of the live data is quite small (climbs to ≈65,000 words and then decreases); GC manages to make this program run without the heap needing expansion. Our current region inferencer ends up placing everything in

Table 2. Space performance comparison (in words resp. thousands of words)

	queens	dnamatch	rdnamatch	browse	rbrowse	serial	qsort	nreverse
Regions –W	196	15,751,504	85,259	1,434,696	2,548,822	2,085 k	6,179 k	20 k
Regions	196	10,471,636	76,111	1,439,576	285,546	500 k	500 k	20 k
WAM	379	194,530	194,533	391,115	391,147	1,571 k	3,144 k	195 k
Allocations	556,324	15,751,504	15,756,504	7,744,666	8,858,792	3,028 k	6,179 k	25,015 k

one region (cf. Table 2). As a result, it needs an enormous amount of space, and gets penalized in execution time as well. The region-friendly version (**rdnamatch**) exhibits a much better space behavior which however comes at a time cost.

The situation is somehow reversed in the **[r]browse** set: **rbrowse** runs faster than **browse** (the region-improved algorithm is inherently faster than the original), and region-based execution is competitive in time with that based on WAM. Moreover, without support for WAM's internal variables, the region-based system results in the program having significantly lower space needs.

Region-based execution is a winner for both **serial** and **qsort**: By avoiding the cost of garbage collection, the total execution times are better by 10% on average (and up to 30% if one compares the –W region system with WAM using a sliding collector). Data in Table 2 clearly show that the space-efficiency of the WAM is a myth. With the most space-efficient region annotation, a region-based system can run in three (**serial**) to six times (**qsort**) less space than the WAM. Interestingly enough, space economy does not seem to pay much in today's machines. The performance of the memory subsystem is apparently quite good in our machine: **qsort** under –W runs faster than without even though it allocates 12 times more space without ever bothering about its deallocation. However, in all machines there is clearly a limit to such space-recklessness.

Finally, we examine the familiar **nreverse** program reversing a list of 5,000 integers. This is a case where region-based execution has the advantage that the size of the used memory is never more than twice the size of the live data (i.e., is linear to the size of the input list). On the other hand, WAM-based execution requires garbage collection and this penalizes execution times. Moreover, note that region-based execution is even faster than WAM-based execution even *excluding* time for GC. This is because the region area fits within the processor's cache (whereas WAM+GC would waste large amounts of time to achieve this).

8 Concluding Remarks

This paper investigated an alternative to WAM's heap allocation- and garbage colection-based memory management: region-based memory management. In particular, we presented a complete scheme for adding region support to the WAM and reported on the performance of the resulting system. In short, our conclusion is that region-based execution is competitive with garbage collected ones, and often offers significantly better time and/or space behavior.

This does not necessarily imply that Prolog systems must abandon the WAM framework. It simply means that alternatives for memory management of logic

programming languages do exist, they can be nicely and tightly integrated in a WAM (or WAM-like) environment, and their performance characteristics can be extremely attractive. We hold that because of this, these memory management schemes, and regions in particular, should be investigated further.

This paper focussed on the abstract machine extensions that regions require: memory architecture and instruction set additions. An orthogonal issue is that of the static analyses that will guide the compiler in generating these new instructions. Although our proposal is not tied to some concrete region inference method, it is clear that the effectiveness of such an analysis also depends on the characteristics of the language that is being analyzed. Adapting our region inferencer so that it is more tailored to Prolog is one direction for future work. Another, perhaps easier to pursue, is to explore region-based memory management in the context of logic programming languages where the concept of static analyses that infer modes and types is not so foreign. Languages such as Ciao Prolog, Mercury, or HAL seem particularly suited for this endeavor.

References

1. Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In Y. Bekkers and J. Cohen, eds, *Proceedings of IWMM'92*, number 637 in LNCS, pages 82–102. Springer-Verlag, Sept. 1992. 164
2. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd *et al*, eds, *Proceedings of CL-2000*, number 1861 in LNAI, pages 1240–1254. Springer, July 2000. 171, 175
3. B. Demoen and K. Sagonas. Heap memory management in Prolog with tabling: Practice and experience. *J. of Functional and Logic Programm.*, 2001(9):1–56, Oct. 2001. 164, 175
4. F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of PPDP 2001*, pages 175–186. ACM Press, Sept. 2001. 165, 168, 169, 175
5. H. Makholm. Region-based memory management in Prolog. Master's thesis, University of Copenhagen, 2000. 166, 171
6. H. Makholm. A region-based memory manager for Prolog. In *Proceedings of ISMM 2000*, pages 25–34. ACM Press, 2000. 164, 166, 170, 171, 176
7. N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. In Codognet, ed., *Proceedings of the 17th ICLP*, number 2237 in LNCS, pages 105–119. Springer, 2001. 163
8. A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure dataflow analysis for Prolog. *ACM Trans. Prog. Lang. Syst.*, 16(2):205–258, Mar. 1994. 163
9. V. Santos Costa. Optimising bytecode emulation for Prolog. In G. Nadathur, ed., *Proceedings of PPDP'99*, number 1702 in LNCS, pages 261–267. Springer, Sept./Oct. 1999. 175
10. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb. 1997. 163, 165
11. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U. S. A., Oct. 1983. 164

A Different Look at Garbage Collection for the WAM

Bart Demoen

Dept. of Computer Science, K.U.Leuven, Belgium

Abstract. A non-algorithmic approach to garbage collection for the WAM heap is developed. A set of garbage collections compatible with the WAM is specified in two steps: the first step makes the useful data for each continuation private and ensures that only useful terms survive garbage collection. The second step completes garbage collection by extending the intuitive notion of folding of identical structures. The role of the trail in the folding process is crucial and it is shown for the ordinary WAM trail as well as for a value trail. New and unexpected opportunities for recovering memory are discovered to be compatible with this view of garbage collection. This approach leads to better understanding of the usefulness logic in the WAM, it is a good start for the formal specification of the garbage collection process and it shows a potential for new compile time analyses that can improve run time memory management. Choice point trimming is used as a vehicle to show selective liveness of data, so its relation to the more common stack maps is established.

1 Introduction

We assume knowledge of Prolog and its implementation, in particular the WAM [20]: see [1] for a good introduction. Some understanding of garbage collection in general (see for instance [12]) and Prolog in particular (see [4]) is also assumed.

We will use the name *heap* for the area in which the WAM allocates its structured terms: others prefer the name global stack. We will also use a double stack model, i.e. as if the environment stack (which we also name the local stack) and the choice point stack are separate. In the figures, stacks will grow downwards. A (heap) segment is a part of the heap delimited by two consecutive choice points.

After [3,4] it seemed as if garbage collection for Prolog and the WAM was understood completely. However, later work like [5,11,13,7,8,6] shows that there is still quite a bit of variation possible in the basic algorithms, and also that some fundamental issues are not so clear cut. In [9] we made a start with better understanding by using Prolog for specifying Morris' algorithm [15] which is a commonly used algorithm for garbage collection of the WAM heap. This specification was high-level and executable. But the situation was very unsatisfactory, especially because other memory management actions – early reset and variable shunting in particular – are rather awkward to integrate in the (executable) specification, even though they are relatively easy to implement

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 179–193, 2002.
© Springer-Verlag Berlin Heidelberg 2002

in an imperative language. The reason seems that garbage collection is mainly understood through the algorithms implementing it. Moreover, while garbage collection should be concerned with keeping the useful *data* alive for the future execution, garbage collection algorithms are usually formulated in terms of pointers and reachability of locations. The notion of weak pointers[1] can alleviate this situation a bit, but few – if any – people think of the WAM trail as a stack of weak pointers.

Still, intuitively it is clear what any garbage collection process attempts to achieve: loosely speaking, it minimizes the total amount of space needed while respecting some constraints, so that the computation can proceed *undisturbed* – see Section 2 for more details on the latter notion. So basically garbage collection is a minimization problem and in the literature, one finds regularly the informal claim that a particular algorithm is optimal, but without stating the constraints. Algorithms seem to hinder a better understanding of these issues, so we will keep away from them. We will describe a set of garbage collections compatible with the WAM in two steps: in Section 3 we describe how every continuation can be given explicitly its useful data and no more. This process involves a potential duplication of data, it is non-deterministic and it covers three known but non-trivial actions, two of which are commonly implemented. Section 4 then describes a set of transformations that are consistent with the view that garbage collection must minimize the space and some trade-offs become explicit. Sections 5 and 6 show interesting and unexpected consequences of the two step view on garbage collection.

These two steps are first explained within the context of binary programs and the usual WAM trailing. The former restriction is lifted in Section 8 while the latter is extended to include value trailing in Section 7. Section 9 explains the relationship between choice point trimming and stack maps.

2 What is Garbage Collection for the WAM?

For simplicity, we will make a series of assumptions: these will make it possible to focus on the essence, but do not restrict in any way the generality of the discussion. The first assumption is that we deal only with binary Prolog programs as in [19]. When a binary program is executed in the WAM, the local stack remains empty, so we can exclude it from our discussion: in Section 8 we will shortly describe how we propose to deal with the environment stack.

In most Prolog systems, garbage collection can occur only at specific points in the execution, say during the execution of a *call* instruction, at the entry point of a predicate or at an equivalently safe point during the execution of a built-in predicate. We will assume here that garbage collection is triggered at the moment of calling a predicate, and since we are dealing with binary Prolog programs, this means during the *execute* instruction: at that point, the heap, trail stack, choice point stack and argument registers contain all the usable information about the

[1] Pointers that do not keep data alive

future continuations of the machine. Indeed, in its simplest form, the game of memory management does not allow to peek into the program code at garbage collection time to make memory management decisions.

One usually distinguishes the forward continuation from the failure continuations each of which is a (potential) forward continuation after the execution has backtracked one or more times. This distinction is not needed here and we will talk about a sequence of continuations $C_1, C_2, ...$ which are ordered by age: C_i is executed earlier than C_{i+1} – barring the execution of the Prolog !/0 (which we cannot take into account because of the rules of the game).

Many implementations start a garbage collection by pushing an extra choice point in which the current argument registers are saved, so that the treatment becomes more uniform. We will do the same. This simplifies matters because the argument registers are no longer relevant at the moment of garbage collection.

In this way, the stacks (heap, trail, choice points) determine together with the program completely the future of the computation and we denote them by WAM_{bef}. Garbage collection transforms this machine state in a WAM_{aft} which has the *same future* as WAM_{bef}. Because of space limitations, we remain vague about what *same future* exactly means. But it is clear that none of the code pointers can be adapted and that WAM_{aft} has the same number of choice points as WAM_{bef} (or more). In fact, as garbage collection should not inspect the program code, i.e. garbage collection must base its actions only on WAM_{bef}[2], WAM_{aft} must be independent of the program that is executing.

The *same future* condition on WAM_{aft} ensures correctness, but usually one is interested in at least one more property of the garbage collection process:

WAM_{aft} must not be ***larger*** then WAM_{bef}

This assumes that we know what *larger* means, but we will not try to establish this, as we are very little concerned with the above requirement here. Depending on other requirements or choices, we can also impose

- the order (in memory) of variables remains the same
- the size of the heap on backtracking starting from WAM_{aft} is never larger than when starting from WAM_{bef}

The former one excludes a plain copying collector. The latter imposes some form of segment preservation.

We could also go for seemingly more weird requirements:

- no reference chain becomes longer (see [7] for more about this)
- the trail does not become larger (Example 2 shows that this is unreasonable)
- the WAM_{aft} heap must be minimal immediately after a garbage collection
- the representation of WAM_{aft} must be minimal at the start of each continuation that existed at the moment of garbage collection

[2] This can be relaxed to allow inspection of compiler generated stack maps.

The first two are actually enforced by most collectors and many people might believe the last two, but the last one cannot be combined with preserving the order of variables because it takes rejuvenation of future garbage to achieve it.

Once we have reformulated the garbage collection process in two steps, it will become intuitively clear that finding a minimal WAM_{aft} is an NP-complete problem. Still, garbage collectors typically run linearly in the size of at most the heap and the root set. It means that garbage collectors for the WAM are *just* implementing a (very) good heuristic. The point might appear academic, but it is instructive to find out what current garbage collectors miss. Doing this in the setting of (new) algorithms limits the imagination and that is a good reason to keep away from algorithms.

3 Privatizing Useful Data

The first step in our approach to garbage collection consists of making all useful data private to each continuation: this transforms a WAM_{bef} into an intermediate WAM_{int}. Let WAM_{bef} have N continuations, meaning there are also N choice points (including the extra one) and N trail segments. Then this transformation can be loosely specified as follows:

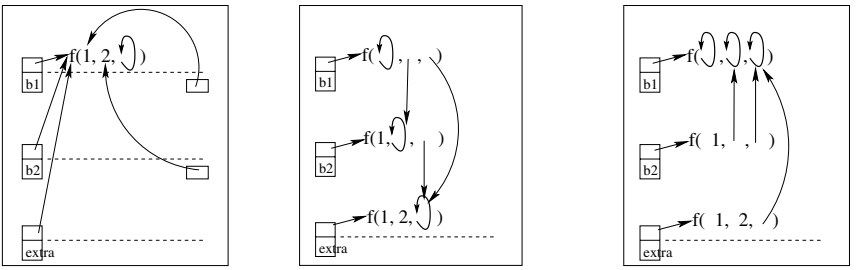
for i **= 1 upto** N **do**
- copy the terms referred to from choice point CP_i**, and**
relocate the references in CP_i
- untrail from trail segment TR_i

We need to clarify what *copy* exactly means and also what happens with the heap and trail pointers in the choice points. We come back to these points later and rely for now on some figures to make things clear. Consider the program

```
run :- A = f(X,Y,Z), b1(A).
b1(A) :- A = f(1,_,_), b2(A).      b2(A) :- A = f(_,2,_), ***(A).
b1(A) :- use(A).                   b2(A) :- use(A).
```

and let the query be $? - run$. We have indicated the moment of doing the transformation from WAM_{bef} to WAM_{int} by the *** in the code: during the *execute* instruction of the first clause of *b2/1*. This notation will be used later on as well, possibly with a numeric index. The predicate *use/1* serves the purpose of keeping its argument alive.

Figure 1(a) shows the WAM_{bef}: the dotted horizontal lines delimit segments; to the left are the choice points for b1 and b2 and the extra choice point we pushed for convenience. The trail entries are at the right of that subfigure. Figure 1(b) shows a WAM_{int}: note that the variables shared by different continuations, remain shared after the transformation. But other terms are disconnected. Note that the trail in 1(b) is empty and each continuation could start executing by reinstalling the saved arguments from its choice point: execution of a continuation will of course need the trail again, but no part of the trail before the transformation. Still, because of the shared variables the continuations cannot execute independently.



(a) WAM_{bef} (b) One possible WAM_{int} (c) Another WAM_{int}

Fig. 1. Illustration of the transformation

We put the following additional restrictions on the copy

- reference chains cannot end in a bound cell
- reference chains ending in a self-reference – the WAM representation of an unbound variable, also named an *undef* – cannot be longer than one
- if a self-reference cell is referred to from within a structure, it must be part of a structure.

While these restrictions are not essential, they make it easier to deal with subsequent transformations. They imply that the WAM_{int} in Figure 2 is not acceptable because it contains an undef which is referred to from inside a structured term, but which is not a cell of a structured term itself. And it contains a reference to the integer 2. On the other hand, Figure 1(c) is also a good alternative WAM_{int}. It means that the transformation from WAM_{bef} to WAM_{int} is non-deterministic. In the figures with a WAM_{int}, we have omitted the segmentation of the (new) heap and we have indicated only its top: we still need to make a decision on where the heap pointers from the choice points need to point to. Since the WAM reclaims heap on backtracking, the only requirement correctness imposes is that the heap pointer of a choice point CP_i points at least as high in the heap as any of terms choice point CP_i refers (indirectly) to. We will not pay any more attention to this issue. The trail pointers in the choice points all point to the start of the trail, since the trail is empty.

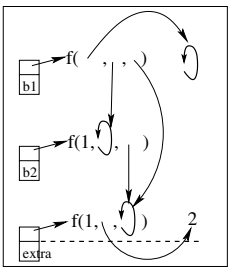


Fig. 2. Bad WAM_{int}

The above informal description should be intuitively clear by now and it can be coded quite easily. But that would be besides the point because it is by no means the intention to produce a WAM_{int} as an intermediate stage in a real garbage collection.

Lest the reader be misled: two different WAM_{bef} can result in the same WAM_{int}. It is not too difficult to construct an example.

The total amount of memory needed after the transformation can be larger than what was in use just before making it. In fact, in the example in Figure 1(b) it has doubled. So it is not immediately clear what the use is of this step in garbage collection. In [3] a similar forking of machines at choice points was considered in the conclusion, for the sake of making an (informal) argument in what sense the there proposed garbage collection schema is optimal.

Even though the transformation does not diminish the memory requirements, a WAM_{int} exhibits a number of interesting properties. In particular: variable shunting, early reset and rejuvenation of future garbage is dealt with optimally. And of course cleaning up the trail is achieved as well. We will discuss in a bit more detail the first three issues.

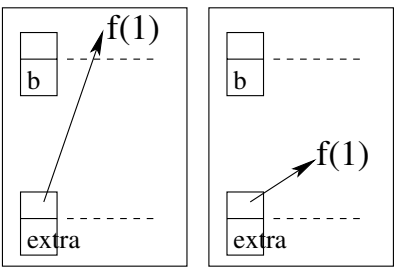
But first note the nicest property of a WAM_{int}

pointer reachability coincides completely with usefulness

Rejuvenation of Future Garbage In [11] it was shown how a copying phase from old to new root pointers moves terms to the oldest segment for which they are useful, i.e. terms can move to a strictly younger segment than the one they were born in. In a WAM_{int} every piece of useful data has been duplicated for each continuation it was useful for, so a natural division of this data into segments, achieves the same – or maybe even a better – migration as in [11]. The following code and Figure 3 exemplify rejuvenation of future garbage:

```
run :- X = f(1), b(X).
b(X) :- ***(X).
b(_) :- ...
```

During the transformation, the term $f(1)$ migrates from the oldest heap segment to the youngest one. This is potentially advantageous for the memory management during the future computation because the term will be popped off the heap by backtracking earlier. Note that rejuvenation of future garbage relies on a compiler optimization named *choice point trimming* (see e.g. [16] and references therein) which is not implemented by many systems. For the example it means that the choice point for b/1 does **not** contain a (direct or indirect) reference to the term $f(1)$, as can be seen in Figure 3(a).



(a) WAM_{bef} (b) WAM_{int}

Fig. 3. Rejuvenated future garbage

Variable Shunting Variable shunting consists in making reference chains shorter and was introduced by [18]. It is clear that after performing the privatizing transformation, no reference chain has length more than one and even this can only happen for free variables. It is not difficult to see that the shunting achieved in a WAM_{int} is perfect and truly optimal. It is strictly stronger

than [18] and the variant of shunting described in [6]. Also the tiny extension to [18] introduced in [7] is covered. So, the known algorithms for shunting cannot be considered optimal, unless one takes into account the whole set of constraints one wants garbage collection to satisfy. In fact, shunting is only perfect, if one allows duplication of terms. This trade-off is now explicit.

It is possible that in some continuations a chain that used to have length zero has length one after the transformation: some chains can be made shorter at the expense of others. This does however not violate the basic principle of variable shunting. Note that this was already described in [7] and also easy-shunting [6] can have this effect.

Early Reset Finally, also early reset is dealt with already: everything that could be reset, has been reset. This follows immediately from the fact that the trail is empty.

4 Sharing the Private Data or Folding Identical Structures

The total amount of memory needed after performing the first step can be larger than what was in use just before taking it. The reason is of course that sharing of terms between continuations (and possibly even within one continuation) is lost. In the next step we will (re-)introduce sharing among terms. This idea has been expressed already in [3]: it seems to resemble hash-consing as in [2], but is not further elaborated on in [3]. Later, in the acknowledgement of [18], it is commented on by the sentence: *It remains to be seen, however, what we meant by* ***folding identical structures***. This seems to indicate that the authors of [18] did not have a clear idea about this notion.

Hash-consing can be performed at garbage collection time as described for instance in [2] for ML. In order to take full advantage of hash-consing by the collector, the mutator must be adapted. The results in [2] are however not completely in favour of the technique. Moreover, in the context of a logic language, i.e. with backtracking and logical variables – hash-consing takes on a different dimension: given the analogy between lazy data structures and logical variables, it is no coincidence that hash-consing was studied in a strict language. Furthermore, the phrase *identical structures* in [18] suggests an equivalence relation, which also hash-consing relies on. We will show however that the right notion – until we introduce value trailing – is subsumption, which is not an equivalence relation.

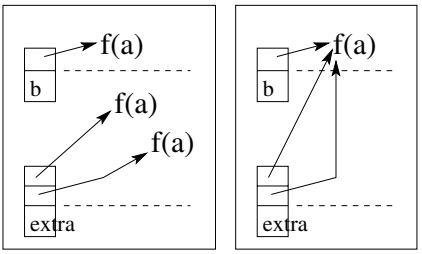
We demonstrate this by some examples: the Prolog code in the examples will contain manifest terms, but one should consider these as if undetectable by the compiler and constructed at run time.

Example 1. For the code

```
run :- X = f(a), b(X).          b(X) :- Y = f(a), ***(X,Y).
                                b(X).
```

The WAM_{int} is shown in Figure 4(a); the result of folding identical terms is WAM_{aft} in Figure 4(b).

Example 1 shows the folding of identical structures that were accessible from different continuations, into one structure. Identical has the usual meaning of the Prolog built-in predicate *==/2*. After such folding, the term must belong to a segment at least as old as the oldest continuation involved in the folding: indeed, in the example only two continuations are involved, but in general, any number of continuations (even just one) could be involved. This kind of folding is similar to what hash-consing achieves. The example shows such folding for ground terms, but it is easily extended to identical (as for == /2) non-ground terms. But we can also fold terms that are not ==/2-identical, as shown in Figure 5.

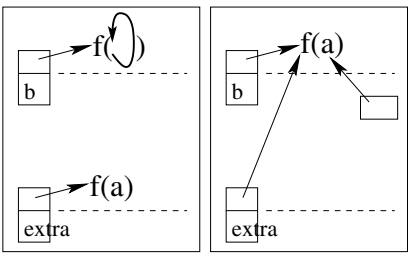


(a) WAM_{int} (b) WAM_{aft}

Fig. 4. Folding identical structures

Example 2. For the WAM_{int} in Figure 5(a), a possible WAM_{aft} is shown in Figure 5(b).

The non-identical structures in Example 2 can be folded, because the two terms were accessible from *different* continuations and the *older* term *subsumes* the newer term: the latter is reflected by the fact that we have introduced a trail entry. Example 2 is very important, as it shows that (1) the naive notion of *identical* structures is insufficient to allow some useful kinds of folding; (2) the introduction of WAM trail entries gives a powerful means to extend the applicability of folding. The naive notion of identical structures would entail that we are looking for an equivalence relation. However, consider the following situation: let $T_1 = f(a, b)$, $T_2 = f(a, _)$ and $T_3 = f(_, b)$, and let T_1 be accessible from continuation C_1 and the other two from C_2. Then T_1 and T_2 can be folded (by introducing a trail entry) or T_1 and T_3 can be folded (also introducing a trail entry). But T_2 and T_3 can not be folded because terms belonging to the same continuation, must be ==/2-identical for allowing folding. Only when terms belong to different continuations, it is enough that one term subsumes the other.



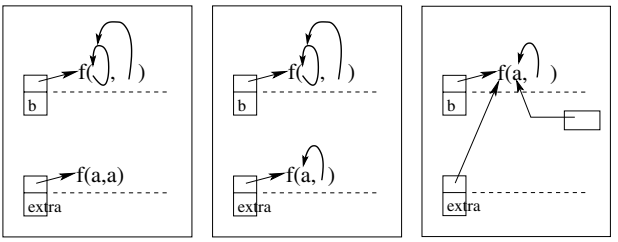
(a) WAM_{int} (b) WAM_{aft}

Fig. 5. Folding non-identical structures

We also note that in Figure 5(b) the space requirements of the machine are now minimal, in the sense that all heap segment lines are minimal.

We can now also fold a variable (a self-reference) in one continuation with a different variable in another continuation, because clearly, one subsumes the other. Such var-var folding does not even need to introduce a trail entry.

Finally, note that folding identical terms can introduce reference chains that end in a bound cell: folding destroys optimal shunting. But it must be allowed or WAM_{aft} cannot be minimal. The succession of figures in Figure 6 shows this: in Figure 6(b) a reference chain is introduced by folding a with a; in Figure 6(c), a final folding introduces a trail entry.



(a) WAM_{int_1} (b) WAM_{int_2} (c) WAM_{aft}

Fig. 6. Two foldings to finish the garbage collection

It is very well possible that the optimal WAM_{aft} is identical to WAM_{bef}, so the question is: what did we gain in this two step process? First of all, we gained some understanding in the interaction of different actions taken by garbage collectors. And further, the explicit possibility to introduce trail entries allows for some freedom as to *which* segment the trail entry is introduced in. This is the subject of Section 5.

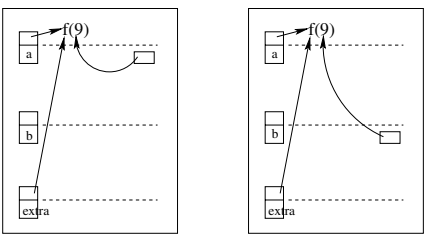
5 Rejuvenating Trail Entries

Example 3. Consider Figure 7 in combination with the program

```
run :- a(f(X)).
a(Z) :- Z = f(9), b(Z).
a(Z) :- use(Z).

b(Z) :- ***(Z).
b(_) :- ...
```

Since the term $f(9)$ is not used in the second clause of b/1, the trail entry that trails the binding of X to the number 9, can be moved to the segment younger than the choice point for b/1.
The example shows that during garbage collection *future* garbage can be made



(a) WAM_{bef} (b)WAM_{aft}

Fig. 7. A trail entry became younger

available for collection earlier by manipulating the trail. This would be useful if instead of the number 9, a large data structure were bound to X and if during the execution of the second clause of b/1, another garbage collection would occur.

One might think that the same effect can be obtained by moving (at the source level) the unification $Z = f(9)$ down to the body of b/1: however, in general this might lead to multiple executions of the unification and also, the above program is of course *too manifest*.

Note that X in Example 3 is not subject to early reset. The example shows in fact that early reset is *just* a special case of a trail entry being rejuvenated to the segment that is younger than the extra choice point, in which case it can be used for resetting immediately.

One challenge is to generalize example 3 and characterise the situation in terms of a marking phase so that garbage collection can really use it. Static analysis can derive enough information to help the garbage collector achieve such rejuvenation.

6 Beyond Folding Identical Structures

In response to a challenge by Marleen D'Hondt [3] in 1988, we produced a picture with the heap layout as in Figure 8 – the L or S in some cells, means a list- or structure-tagged pointer. Such a heap layout could have been produced by the execution of the query $? - X = [a \mid b], Y = f(a, b)$. but then the compiler would have to generate a rather unusual instruction sequence like ***get_struct Y, f/2 ; get_list X ; unify_atom a ; unify_atom b***. This can be seen as an application of common subexpression elimination pushed to the absurd or as the result of unusual folding of identical structures. In the example, the common subexpression elimination could be performed at compile time because the common subexpressions were manifest. In general, such a common subexpression elimination can only be performed at run time and garbage collection time is a good opportunity because it has a global view on the data in the program. It therefore fits in well with the view of finding a minimal representation for the set of useful terms. It seems however beyond the original intention of folding identical structures.

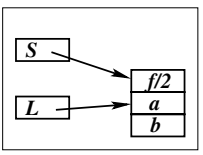


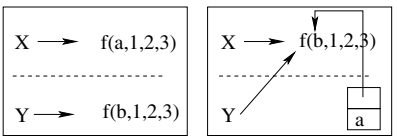
Fig. 8. Weird heap layout

7 The Value Trail and Folding Structures

Many Prolog implementations these days employ a value trail: it allows to destructively update a data structure on the heap, so that on backtracking the old value can be re-installed. A value trail is also used for re-installing forward bindings, as in the XSB execution model ([17]), and in parallel execution mechanisms like [14], and also in CLP systems. A value trail is definitely

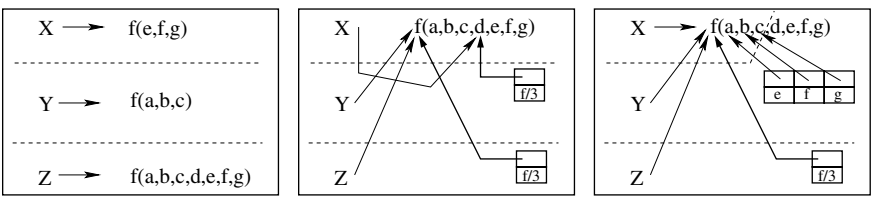
[3] One of the ProLog-by-BIM team members

popular enough to warrant attention. Moreover, it gives a new dimension to the possibility that different processes can have a different view on the same location. Indeed, multiple values in one location occur in WAM for instance when a heap cell is bound to an integer say and it is trailed: in forward execution, the cell contains the integer. In the alternative forward execution – that is, after some backtracking – the cell contains again an undef. So in plain WAM, there are at most two views on a cell: bound or undef. With a value trail, the number of views is bound only by the number of continuations – note that for instance [16] describes a technique for avoiding multiple value trailing of the same location within the same segment. Introducing value trail entries during garbage collection is quite powerful at minimizing the heap consumption. Figures 9 shows folding by introducing a *normal* value trail entry: the value is something that can be the end of a reference chain. Figure 10 shows how also unusual value trail entries can be introduced and that the minimization problem of garbage collection becomes even more complicated in the presence of the value trail. Two alternatives are shown: the one Figure 10(b) is minimal just after the transformation, but after backtracking twice, the one in Figure 10(c) is.



(a) WAM_{int} (b) WAM_{aft}

Fig. 9. An ordinary value trail entry



(a) WAM_{int} (b) WAM_{aft_1} (c) WAM_{aft_2}

Fig. 10. Introducing unusual value trail entries

8 Dealing with the Local Stack

Up to now, we restricted the discussion to binary programs for simplicity. Indeed, at first sight the environments complicate the picture a lot. However, using binarization of a non-binary program, applying the ideas above to the binary version and translating back to environments, will give insight. The crucial realization is the relation between the term on the heap used for the forward continuation and an environment. The following example shows this.

```
% original clause                % binary clause
a(X,Y) :- b(X,Z), c(Z,Y).       a(X,Y,Cont) :- b(X,Z,c(Z,Y,Cont)).
```

When executing a/2, its environment contains the variables Z and Y, because they have to survive the call to b/2. Analogously, the variables Z and Y survive the execution of b/3, because they are in the continuation term c(Z,Y,Cont). One can easily see that the continuation term is never subject to choice point trimming: indeed, binarization makes sure that the incoming continuation is always used in the body. This means that up to the last clause for b/3 (included) the continuation term is live and this corresponds to the stack map that is associated to the call to c/2 in the body of a/2, which keeps exactly Z and Y alive – together with the bare environment frame – as long as b/2 has not finished. The main difference between the continuation terms in the binary clause and the environment for the original clause, is that continuation terms can never have a variable that is not initialized as this is not compatible with the WAM.

The following example indeed shows that at the moment when b/2 is executing, only B and D need to be kept alive, while in the binary form, the corresponding continuation term c(B,C, d(C,D, Cont)) also keeps C alive.

original clause: **a(A,D) :- b(A,B), c(B,C), d(C,D).**
binary clause: **a(A,D,Cont) :- b(A,B, c(B,C, d(C,D,Cont))).**

Even despite this difference, the analogy is strong enough to make clear that environments act like heap terms, in that they can in the first step (Section 3) be replicated for different continuations and with a different set of active variables. In fact, one should imagine this first step taking into account any stack maps. In step two, environments can be collapsed, possibly with the introduction of trail entries. Collapsing was not always beneficial for heap terms, but in the case of stack frames it is: first note that even if there are no live variables in a stack frame, its size is two because the continuation pointer and the previous environment pointer are always present. The next notable point is that variables cannot be moved in an environment (in an attempt to compact it), because WAM code was generated for fixed locations. It follows that two replicas of the same original environment will occupy always less space when collapsed because one needs to introduce trail cells for at most the variable slots which are common to both frames. Whether collapsing frames while introducing trail cells is also good for performance, is of course a different matter.

Finally, note that in the process of steps 1 and 2, one can trim frames: in the WAM this is named environment trimming and implemented as a mutator action on the basis of compile time information which is a crude approximation of stack maps. It is clear that environment trimming could be left to the collector.

9 Stack Maps and Choice Point Trimming

Choice point trimming (CPT) is not commonly implemented, possibly because in an emulator context it can lead to more emulator cycles. On the other hand, stack maps are used by several systems. And the combination of environment trimming with initializing environment variables as in SICStus Prolog is roughly equivalent to maintaining stack maps at runtime, instead of constructing them

at compile time. Since the previous sections rely on CPT, it is worth showing its relation with the more commonly used technique of stack maps. This also fits the re-introduction of the local stack, as (usually) stack maps apply to the environment variables.

Consider the following two equivalent definitions of the predicate p/1:

```
% definition 1                  % definition 2
p(X) :- a(X), b(X).             p(X) :- a(X), (c(X), ***2 ; ... ).

b(X) :- c(X), ***1.
b(_) :- ...
```

Assume the query is $?-p(< Term >)$. At program point ***1 and assuming CPT, the $< Term >$ is dead, because the choice point of b/1 does not contain a reference to this value.

On the other hand, at program point ***2 $< Term >$, the value of X, is dead and this is reflected in the stack map associated to this program point.

This shows that choice point trimming has a similar effect as stack maps in an implementation which compiles disjunction in-line.

While the above example shows the relation between choice point trimming and stack maps in a non-deterministic context (for the stack maps), the report [10] shows that the relation also exist in a deterministic context, i.e. an implementation that performs choice point trimming can be used to mimic stack maps.

Even though few systems implement CPT, we feel that using CPT in the previous sections was justified, exactly because of the relation between CPT and the more common stack maps.

10 Conclusion

In the past the understanding of usefulness logic was often linked directly to algorithms for exploiting this understanding in a garbage collection context. Since the attention then shifts naturally to efficiency and practicality, it is no surprise that some consequences of the usefulness logic are never discovered. Also, one is inclined to preserve WAM invariants and since these have not been studied fully, one tends to be overly conservative. For instance, efficiency considerations seem to prevent thinking about making changes to the trail, except for compacting it. Still, other changes to the trail are possible and compatible with the strict minimum of WAM invariants and, as shown in Sections 5 and 7, they can improve overall memory management. Also rejuvenation of future garbage seems to break a WAM invariant: *terms can reside in a heap segment that is not younger than the segment they were created in.* But it is a false invariant and not worth enforcing. The abundant absence of some compiler optimizations (like choice point trimming) re-enforces this false intuition for WAM invariants.

By breaking away from these traditional views, we get both a better understanding of the usefulness logic of the WAM and of the chances for better

memory management we have missed in the past. The future will teach whether some of these are worth considering for an implementation.

The value of this work lies in offering a clearer understanding of what usefulness logic for the WAM is and in providing a framework which seems to cover all existing techniques which up to now seemed ad hoc and unrelated, and even a few novel ones which are equally ad hoc and unrelated. We believe we are now in a better position to reason about the correctness of a specific garbage collection process by making it fit into the framework. Moreover, our interpretation of what *folding identical structures* could mean allows definitely to recover more garbage than the one in [3] and goes beyond plain hash-consing at garbage collection time as proposed in [2]. Rejuvenation of trail entries was discovered while exploring the frame work, as it fits in neatly with the constraints.

None of the garbage collection opportunities described here might turn out to be practical, but we feel we have reached our goal: to point out some consequences of a clearer definition of garbage collection for the WAM and the fact that they are unexpected. This means that the understanding of garbage collection lags behind actual implementation. Any formal specification of garbage collection for Prolog must take into account these issues at the risk of not covering the issue in full. It is now also clear that [9] was just a brute force approach to the original intention of finding executable specifications of garbage collection in the hope to find classical algorithms.

Backtracking and the trail lead to memory cells whose contents depend on the continuation. Such multiple view issue seems also to exist in other language contexts. It remains to be seen whether any of the above is useful there too.

Acknowledgements

We are grateful to Olivier Ridoux for raising our awareness of the importance of understanding multiple views on the same location. We also thank the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest of Angers, France where part of this work was conducted.

References

1. H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990 See also: http://www.isg.sfu.ca/~hak/documents/wam.html. 179
2. A. W. Appel and M. J. R. Goncalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Feb. 1993. 185, 192
3. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988. 179, 184, 185, 192
4. Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In Y. Bekkers and J. Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 82–102. Springer-Verlag, Sept. 1992. 179

5. J. Bevemyr and T. Lindgren. A simple and efficient copying garbage collector for Prolog. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 88–101. Springer-Verlag, Sept. 1994. 179
6. L. F. Castro and V. S. Costa. Understanding Memory Management in Prolog Systems. In P. Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming, ICLP'2001*, number 2237 in Lecture Notes in Computer Science, pages 11–26. Springer-Verlag, nov 2001. 179, 185
7. B. Demoen. Early reset and reference counting improve variable shunting in the WAM. Report CW 298, Dept. of Comp. Sc., K. U.Leuven, Belgium, Aug. 2000. 179, 181, 185
8. B. Demoen. Marking in the presence of destructive assignment is suboptimal. Report CW 302, Dept. of Computer Science, K. U.Leuven, Belgium, Oct. 2000. 179
9. B. Demoen. Prolog and abduction 4 writing garbage collectors. In K.-K. Lau, editor, *Pre-Proceedings of Tenth International Workshop on Logic-based Program Syntthesis and Transformation, 2000*, pages 128–135. University of Manchester, 2000. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. 179, 192
10. B. Demoen. A fresh look at garbage collection for Prolog. Report CW 330, Dept. of Computer Science, K. U.Leuven, Belgium, Jan. 2002. 191
11. B. Demoen, G. Engels, and P. Tarau. Segment order preserving copying garbage collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386. ACM Press, Feb. 1996. 179, 184
12. R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management*. John Wiley, 1996 See also http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html. 179
13. X. Li. Efficient memory management in a merged heap/stack Prolog machine. In *Proceedings of the 2nd ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 245–256. ACM Press, 2000. 179
14. E. Lusk, R. Butler, et. al. The aurora or-parallel prolog system. *New Generation Computing*, 7(2,3):243–271, 1990. 188
15. F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, Aug. 1978. 179
16. J. Noyé. *Elagage de contexte, retour arrière superficiel, modifications réversibles et autres: une étude approfondie de la WAM*. PhD thesis, Universite de Rennes I, Nov. 1994. 184, 189
17. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conference*. ACM, 1994. 188
18. D. Sahlin and M. Carlsson. Variable Shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991. 184, 185
19. P. Tarau. Program transformations and wam-support for the compilation of definite metaprograms. In A. Voronkov, editor, *Russian Conference on Logic Programming*, number 592 in Lecture Notes in Artificial Intelligence, pages 462–473, Berlin, Heidelberg, 1992. Springer-Verlag. 180
20. D. H. D. Warren. An abstract Prolog instruction set. Tech. Report 309, SRI, 1983. 179

Copying Garbage Collection for the WAM: To Mark or Not to Mark?

Bart Demoen[1], Phuong-Lan Nguyen[2], and Ruben Vandeginste[1]

[1] Dept. of Computer Science, K.U.Leuven, Belgium
[2] Institut de Mathématiques Appliquées, UCO, Angers, France

Abstract. Garbage collection by copying is becoming more and more popular for Prolog. Copying requires a marking phase in order to be safe: safeness means that the to-space is guaranteed not to overflow. However, some systems use a copying garbage collector without marking prior to copying, and instead postpone the copying of potentially unsafe cells. Such systems only collect small portions of the heap and it is not clear whether postponing works while collecting the whole heap. Moreover, it is shown here that postponing does not solve the problem in a fundamental way. Since marking takes time, it is worth studying the tradeoffs involved. These observations have prompted the experimentation with a series of garbage collectors based on copying without marking and without postponing. In particular, variants were implemented that are named *dangerous*, *optimistic* and *cautious* copying which exhibit various degrees of unsafeness. Versions of each have been implemented based on recursive copying as in most implementations of *copy_term/2* and on the Cheney algorithm. Performance on benchmarks suggests that large performance gains can be obtained by skipping the marking phase, that dangerous copying is still relatively safe but can be costly, and that the additional effort of cautious copying over optimistic copying is not worth it. The optimistic collectors based on recursive copying perform best and slightly better than the ones based on Cheney. Cache performance measurements back up the benchmark results.

1 Introduction

We assume knowledge of Prolog and its implementation. For a good introduction to the WAM [18], see [1]. Knowledge of garbage collection in general (see for instance [12]) and Prolog in particular (see [3,2]) is also assumed. We will use three WAM based systems in this paper: SICStus Prolog, Yap and hProlog. The first two are only used for making a short comparison of their garbage collectors to the hProlog one in Section 2. The SICStus Prolog implementation is described in [5]. About Yap one can find implementation details in [8]. hProlog is a successor of dProlog [10] and is available from the first author. hProlog is meant to become a back end to HAL [9].

We will use the following terminology: the *heap* is the area in which the WAM allocates structured terms – elsewhere the term *global stack* is often used

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 194–208, 2002.
© Springer-Verlag Berlin Heidelberg 2002

for this area; the *local stack* is the stack with environments or stack frames in more traditional terminology; the *choice point stack* contains the choice points. The original WAM uses a merged environment/choice point stack as Yap still does. SICStus Prolog and hProlog have separate local and choice point stacks. We will also employ the usual terminology about copying collectors: to-space and from-space, and scan-next pointers from the Cheney algorithm. We will also abuse terminology slightly: when we say *collect a term* we actually mean that the term is rescued by a garbage collection.

Work on the topic of the paper started in [11] with a wider comparison of the implementation of the built-in predicate copy_term/2 and garbage collection. Copying a term with the built-in copy_term/2 and garbage collection are indeed related: both make a duplicate of a (set of) term(s). Of course copying a term and collecting it have different properties: copying a term may shorten all reference chains, while garbage collection must take into account the trailing state of the cells in the chain (see [15] for more details); copy_term/2 must preserve the original term, while garbage collection may destroy it; garbage collection must preserve the sharing between terms (again because of the trailing state) while copy_term/2 optimally should do so, but is only required to preserve sharing of variables. There is also another remarkable difference between the garbage collectors in systems like SICStus Prolog, Yap and hProlog on one hand and their implementation of copy_term/2: their garbage collectors start by marking the live data - the term(s) to be copied by the collector - before the actual copying is done, while copy_term/2 does no prior marking. In SICStus Prolog and Yap, the necessity of marking follows from their choice for Morris' *sliding* algorithm [14] for copying the live data. hProlog however uses the variant of the Cheney algorithm [7] proposed in [4]. The variant involves marking before copying as a means to ensure that garbage collection is safe, i.e. that the heap does not grow during garbage collection. This is ensured by never copying a cell twice. Two issues that have never been settled are studied in this paper:

- how *unsafe* is garbage collection by copying without marking?
- how is performance affected by omitting the marking phase?

In Section 3 we explain in detail that copying without marking – which we name *dangerous* copying – is not safe and how the technique proposed in [4] solves this problem by marking. In the same section, we also show that contrary to common belief, copying without marking can cause the heap to grow almost arbitrarily. On the other hand, we will show that in practice - that is, on a set of benchmarks - this heap growth is not catastrophic at all and one can even limit the amount by which the heap can grow during garbage collection. The performance gain of omitting the marking phase is in some cases substantial.

The basis of a copying garbage collector can be either a recursive algorithm as used in the implementation of copy_term/2 – which takes of course into account the above mentioned differences between copying and collecting a term – or Cheney's algorithm. Since it is not a priori clear which one leads to better results, we have experimented with both. Apart from *dangerous* copying, we have identified

two more degrees of unsafeness and the according variants of the copy algorithm are named *optimistic*, and *cautious*: they are explained in detail in Section 4. So in total we have six garbage collectors that copy terms without marking. In Section 5 we report on their performance on a set of benchmarks, compared to the pre-existing hProlog garbage collector which marks before copying. The benchmarks themselves are discussed in some detail in Section 4. It turns out that optimistic copying makes a very good compromise between safety of garbage collection and performance. We discuss shortly in Section 7 how to recover from an unsafe situation, i.e. how to recover from an overflow of the to-space during garbage collection.

In Section 6 we present cache simulation results for all the variants, which help in explaining and understanding the observed performance differences.

The experimental evaluation was performed on a Pentium III 866MHz with 256Mb RAM. Timings are given in seconds, space measurements in bytes or cells. In Section 2 we used SICStus Prolog 3.8.6, Yap 4.3.20 and hProlog 1.4 throughout.

2 The hProlog Garbage Collector Compared to Others

The main aim of this section is to show that the hProlog garbage collector is of reasonable quality so that any performance improvements to it, are relevant. We therefore compare the garbage collector that existed in hProlog before this research started, with the collectors in SICStus Prolog and Yap.

The collector in hProlog is a rather straightforward implementation of [4] and thus uses a Cheney algorithm [7] with a marking phase. It is non-generational. The collector in SICStus Prolog is an implementation of [2]: it uses a mark&slide algorithm based on Morris [14]. It also implements variable shunting. More importantly, it is 2-generational. The garbage collector of Yap is similar to the one in SICStus Prolog, but it currently uses *easy shunting*: see [6]. It is non-generational.

Because the different systems have different allocation and expansion policies, we have used the following procedure to decide on how much space to give each system:

- for each benchmark SICStus Prolog gets enough heap so that no expansion occurs during the execution of the benchmark,
- Yap gets enough space so that it has exactly the same amount of garbage collections as SICStus Prolog; this is reasonable because SICStus Prolog and Yap both follow the WAM closely in their allocation of free variables, and also in their internal representation of terms,
- hProlog gets exactly the same amount of heap space as SICStus Prolog allocates.

Even when *all things seem equal* a different number of activations of the garbage collector can result, or with an equal number of collections, the results can be skewed, because:

Table 1. The benchmarks and some of their characteristics

benchmark	origin	survival %	survival size (cells)	max trail	max ls	max cp	trigger
emul (*)	[6]	98	16600	172329	3866	9841	overflow
tsp (*)	[13]	98	23300	7895	83	104	overflow
dnamatch (*)	[13]	86	34471	13942	76	104	overflow
chess (**)	[6]	36	1746304	1805065	6276977	5214953	explicit
serial (*)	[13]	34	4042536	5882854	122	104	overflow
boyer (*)	[13]	76	3176103	3985228	463	236	overflow
mqueens	(***)	99.9	5577525	82	129	104	overflow

- with the same heap size, a generational system (like SICStus Prolog) might activate its collector more often than non-generational systems – or put differently: Yap might need less heap space for doing N collections during a benchmark than SICStus Prolog, because SICStus Prolog is generational,
- hProlog allocates all free variables in the heap, so its heap demands are higher and this can result in more frequent collections,
- non-segment-preserving collectors (like the one of hProlog) reclaim less heap space on backtracking and again, this can result in more frequent collections.

When comparing garbage collection algorithms or implementations, the characteristics of the benchmarks should also be taken into account. We have therefore included Table 1 which contains for each benchmark used in the rest of the paper information on its origin (i.e. where we got it from), the percentage of cells that (on average) survive two successive garbage collections, the average number of cells that survive a garbage collection, the maximal trail, local stack and choice point stack size during the execution of the benchmark and finally whether the benchmark triggers garbage collection by running out of heap (overflow) or by calling the built-in predicate *garbage_collect*/1 explicitly. The latter is important because e.g. in SICStus Prolog, such an explicit call to *garbage_collect*/1 does always a major collection, while collections triggered by overflow are often minor ones. The maximal stack sizes give a good idea about the size of the root set on garbage collection: this information can be useful to interpret the results in Section 5. The figures were all obtained with hProlog.

The benchmarks marked (*) are *classical* ones. The program marked (**) was used in [6]; K. Sagonas extended it with one more chess move in order to make the execution more interesting for garbage collection. (***) was written by B. Demoen; it is a naive solution to the M-queens problem of the 7th Prolog Programming Contest; it uses almost exclusively list of lists of integers and atoms.

The very large trail size (for instance for boyer) might seem weird, but hProlog does not tidy the trail on cut as SICStus Prolog and Yap do – and boyer has some pretty non-steadfast predicates. Note also that hProlog has a (non-optimized) value trail.

Table 2. Comparing the mark&Cheney copying collector of hProlog with others

	hProlog	SICStus	Yap
emul	1.83/24.21 - 75	2.04/41.40 - 42	0.74/24.66 - 42
tsp	0.05/24.10 - 8	0.33/54.62 - 6	0.03/44.33 - 6
dnamatch	0.13/3.46 - 15	0.82/5.94 - 15	0.24/4.46 - 15
chess	2.88/12.83 - 3	3.44/23.14 - 3	2.36/12.96 - 3
serial	3.89/11.02 - 3	4.10/18.25 - 3	4.76/13.97 - 3
boyer	4.34/14.04 - 5	2.85/19.54 - 5	3.65/12.09 - 5
mqueens	40.46/97.50 - 41	18.12/141.95 - 38	42.42/86.83 - 38

The benchmarks in Table 1 have been ordered in such a way that the lower the benchmark in the table, the more we think it benefits from generational garbage collection. This is most pronounced for the mqueens benchmark.

Table 2 shows for each system and benchmark the time spent in the collector, in the mutator (i.e. the runtime of the benchmark not including garbage collection time) and the number of garbage collections. No system performs uniformly better nor worse than the other two. Given the maturity of SICStus Prolog and Yap, we feel it is safe to conclude that the hProlog garbage collector is a relevant starting basis for our experiments.

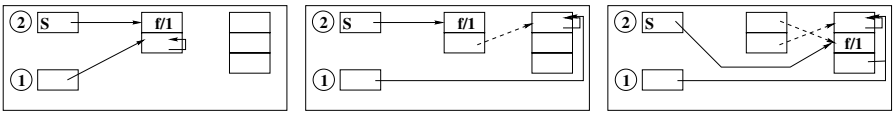
3 Copying without Marking Is Not Safe

As far as we know, the first to observe that copying without prior marking is unsafe in the context of the WAM, was [4]. The reason is that when a root cell that points to an inner cell of a structure is treated by the copying phase before the structure as a whole, this inner cell is copied twice. Figure 1 shows this clearly.

The figures will typically consist of three parts. From left to right, we have a root set, a from-space (or the current heap), and a to-space (the heap after the collection). The cells in the root set point to the from-space, possibly by a tagged pointer: S stands for the STRUCT tag, L for the LIST tag. Dashed arrows will indicate forwarding pointers. The root cells are treated in the order indicated by the numbers in the circles next to the root cells.

Figure 1(a) gives an initial situation in which cell 1 has a reference to a self-reference (the WAM representation of a free variable), and cell 2 has a STRUCT tagged pointer to a functor f/1 which is followed by its argument, which is the self-reference. After root cell 1 is copied, the situation is as in Figure 1(b). Figure 1(c) shows the situation after also root cell 2 has been (recursively) copied. Before the copy was made, the total size of the from-space was two cells; the copy resulted in a from-space with three reachable cells.

The example shows the double copying of a cell containing a self-reference. The same double copying can happen to any cell containing a value that can be in the direct arguments of a structured term or list pair. This means that



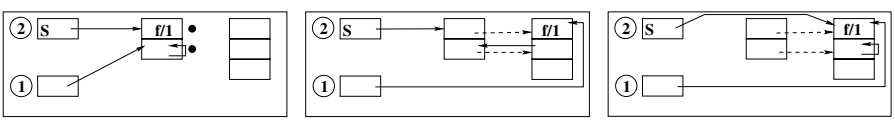
(a) Before copying (b) after copying 1 (c) after copying 2

Fig. 1. Example of double copying of one cell

as soon as a root cell is an non-tagged reference to a (not yet copied) heap cell, copying is unsafe: there is a danger that the to-space (which according to garbage collection practice is as large as the from-space) overflows. Consequently, we name copying without marking *dangerous*. The cell which was copied twice is also named *dangerous*.

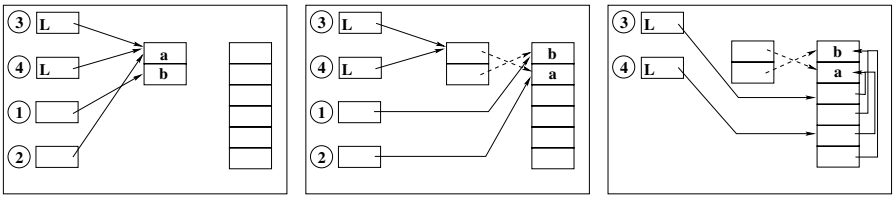
The authors in [4] propose to mark first the live data. When during the copying phase, a root cell points to a not yet copied cell in the heap, the whole block of marked cells is copied as a whole. This is shown in Figure 2: the mark bits are shown by the dots next to a cell. When root cell 1 is treated, the largest block of marked cells containing the cell to which root cell 1 points, is forwarded, resulting in the Figure 2(b). Finally, Figure 2(c) shows the situation at the end of the collection. This procedure guarantees that the copy is not larger than the original.

Since the marking phase costs time (and possibly space), it is worth investigating whether avoiding the marking phase is feasible. It could be argued that double copying is probably very rare, but there is no empirical evidence for that. Another approach is to postpone the (recursive) copying of cells that look dangerous in the sense defined above. This approach is taken in [19] and [13], but only for garbage collections which collect only part of the heap, or even only the part younger than the most recent choice point. Postponing consists in collecting dangerous cells in a set that is treated after all other root cells have been treated. [19,13] both report that the set is always small, but given the fact that the heap is only partially collected, this is not a surprise. Also, if only the most recent heap segment is collected, the notion of dangerous cell is not quite the same as in general, because no cell in the most recent segment is trailed. This implies that reference chains can be shortened, which in itself prevents double copying.



(a) After marking (b) after copying 1 (c) after copying 2

Fig. 2. Using the mark bits to forward a block



(a) Before any copying (b) after copying 1 & 2 (c) after copying 3 & 4

Fig. 3. More than double copying of a list pair

Finally, when collecting all segments, even postponing is not a foolproof recipe against double copying: the order in which postponed cells are treated is indeed crucial. A bad order will lead to double copying. And a good order is impossible to find without some sort of marking phase of at least the postponed cells.

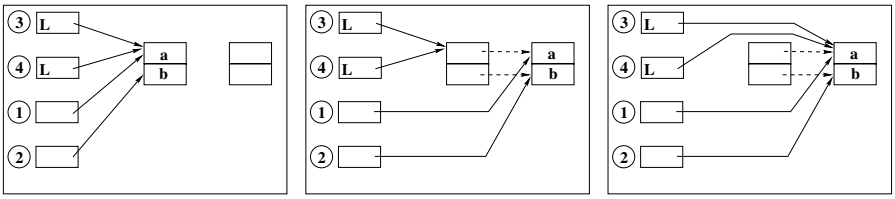
But there is an even worse issue and one that has not been pointed out in literature before: while cells belonging to a (non-list) structured term run the risk of being copied (at most) twice, cells belonging to a list pair might even be copied an arbitrary number of times. Figure 3 shows this.

The initial situation can be seen in Figure 3(a). After the root cells 1 and 2 have been copied, the situation is as in Figure 3(b). At that point root cell 3 is copied: after relocation, the LIST tagged pointer in the root cell must point to a list pair (two consecutive cells on the heap). But the atoms a and b are on the heap in the wrong order. So a new list pair is allocated in the to-space and it contains references to the cells in which a and b are situated. The same happens for root cell 4, resulting in Figure 3(c) (for clarity we have not reproduced cells 1 and 2). This shows that more than double copying can occur for the cells of a list-pair.

Dangerous copying has been advocated by Paul Tarau. It is thus worth noting that the above mentioned problem of multiple (more than double) copying of cells does not occur in the context of *tag-on-data* as in the BinWAM [16], because tag-on-data represents lists in essentially the same way as other binary structures.

4 Copying without Marking Can Be Made Safer

We will present here two techniques for making copying without marking safer: our general point of view is that double copying is probably rare and cannot be avoided without marking, but we should find a practical way to alleviate the problem of more than double copying of list-pairs, or solve it completely. Section 4.1 presents a technique that works well in practice, but can not guarantee to avoid more than double copying. Section 4.2 solves the problem completely, at a small cost.



(a) Before any copying (b) after copying 1 & 2 (c) after copying 3 & 4

Fig. 4. No double copying of a list pair because we were lucky

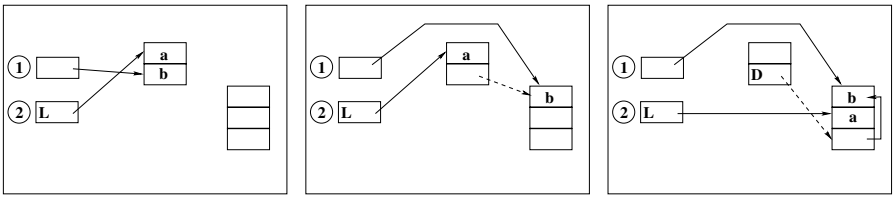
4.1 Optimistic Copying

If the two cells of a list pair by chance are copied to subsequent locations on the heap, it is possible to avoid copying them multiple times. This is illustrated in Figure 4: after root cells 1 and 2 are copied, the cells containing a and b are in locations with addresses that differ by one (unit of heap address). This can be easily checked before subsequent copying starting from a LIST tagged pointer. The resulting copy is shown in Figure 4(c). We name this *optimistic* copying, because the idea is that hopefully the cells of a list are copied as a list-pair together before any duplication occurs.

4.2 Cautious Copying

Optimistic copying does not help when the two list cells are copied at first in the wrong order, or if only one of them was copied before the two cells are copied as a list-pair. Fortunately, even in that situation, we can limit the duplication to two, by using the tag field in the forwarding pointers. Figure 5 exemplifies this. When the second root pointer is treated by the copy routine, it creates a list-pair in the to-space and copies the contents from the list-pair in the from-space to it. It then leaves the usual forwarding pointers, but the one for the second cell of the list-pair is tagged with the tag D which indicates that this cell contained a forwarding pointer before: that previous forwarding pointer can still be retrieved because the D-tagged forwarding pointer points to it. This is shown in Figure 5(c). On following a subsequent list pointer to the list pair, the copy routine can easily recognize this situation as an already copied list-pair, because (after the D-tag has been stripped if needed) the two forwarding pointers in the pair are subsequent. On following any other strong pointer directly to the first cell of the list pair, the D-tagged forwarding pointer is just stripped from its tag and treated as usual or dereferenced: both are correct.

The D-tag is not really used yet in the above. We need it for relocating correctly pointers from the trail to the heap. In order to avoid multiple (more than twice) copying one must sometimes overwrite a forwarding pointer - say f_1 - with a new one - say f_2. Suppose that a trail entry t points to a cell with the overwriting forwarding pointer f_2. To relocate t correctly, it must be made



(a) Before any copying (b) after copying 1 (c) after copying 2

Fig. 5. Limiting the duplication by cautious copying

to point to the location that f_1 points to (if t does not point to a cell with a forwarding pointer, then t can be used for early reset). To recognize the situation, f_2 is tagged with the D-tag, so that by dereferencing it, f_1 can be retrieved. Referring to Figure 5, note that a trail entry t that points to the cell containing b in Figure 5(a), must after the garbage collection point to the cell that contains b in Figure 5(c). Suppose that t is treated between the treatment of root cell 1 and root cell 2, then no problem occurs. If t is treated after root cell 2, the original forwarding pointer has already been overwritten, but the D-tag indicates that an extra dereference is needed. Non-trail root pointers do not need to use the D-tag: the result can be an extra reference in a chain as shown in Figure 5.

The example does of course not cover all possible situations (e.g. the cell with **a** could also have been copied already before a list pointer is found that points to the list pair), but the principle remains the same: the first time two cells are copied as a list pair, any already existing forwarding pointer will be replaced by a D-tagged new forwarding pointer. Such D-tagging is not necessary if the cell cannot be trailed anymore, i.e. it is in the most recent segment of the currently treated continuation.

In the implementation, we have used for D the same tag as for floating point numbers, because that tag occurs least in practice. We name this variant *cautious* copying.

4.3 Final Notes on the Variants

Early reset ([2]) is not mandatory but when it is done, it happens during the marking phase. Early reset requires that marking visits the root set in a particular order, i.e. from new to old continuations. So we make sure that our copying routines visit the root set in the same order which allows us to do the same early resets. The forwarding pointers left by the copying routines now act like the mark bit which is used to check whether a cell has been visited before. In the schema which postpones dangerous cells (like [13,19]), the order of copying becomes different and in order to retain early reset, one should empty the work list (with the postponed cells) before closing a segment with the treatment of its trail. But this entails of course the potential for double copying. The alternative is to give up on early reset as soon as the work list is non-empty.

The adaption of the recursive copying algorithm from copy_term/2 to garbage collection is not difficult: the main issues to take into account are that reference chains cannot be shortened (barring perhaps easy shunting as in [6] which happens during the marking phase) and that the copy can be destructive.

Adapting the Cheney copying algorithm already used in hProlog in combination with marking, to work without marking leaves some room for choices. Cheney's algorithm consists of basically two actions: (1) forward one or more root pointers and, (2) let the *scan* pointer catch up with the *next* pointer. In principle, both actions can be interleaved at will. But we noticed the following:

- forwarding many root pointers before entering phase two leads to more multiple copying of internal cells,
- forwarding root pointers from different segments before entering phase two, conflicts with early reset: one must indeed finish phase two completely for the root pointers of a particular segment, before treating a trail segment during which early reset is performed,
- entering phase two more often leads to a slight performance degradation.

Despite the last point, we have written the Cheney variants so that they go through phase two each time one root pointer has been forwarded.

5 Experimental Results

For each of the implemented variants and each benchmark, Table 3 shows the garbage collection time in seconds, the number of garbage collections, and the total number of bytes collected: the higher the latter number (with an equal number of garbage collections) the more effective the algorithm. We do not give the mutator runtime because we observed that it hardly depends on the collector. We first discuss the space performance and then the time performance.

Table 3. Comparing one mark&copy collector with six plain copying collectors: garbage collection time, # of garbage collections, # of collected bytes

	mark&Cheney	recursive copy without marking			Cheney without marking		
benchmark	hProlog	dangerous	optimistic	cautious	dangerous	optimistic	cautious
emul	1.83	1.88	1.89	1.91	1.90	1.90	1.92
	75	75	75	75	75	75	75
	398222440	398221104	398225852	398225852	398221104	398221104	398221104
tsp	0.05	0.05	0.00	0.02	0.04	0.04	0.03
	8	8	8	8	8	8	8
	42255172	42255100	42255120	42255120	42255100	42255120	42255120
dnamatch	0.13	0.07	0.08	0.10	0.12	0.08	0.09
	15	15	15	15	15	15	15
	78565940	78564528	78565684	78565684	78564528	78565684	78565684
chess	2.88	1.77	1.67	1.74	2.09	1.93	2.02
	3	3	3	3	3	3	3
	14646248	10986964	14427476	14529664	10986964	14646244	14646244
serial	3.89	2.72	2.73	2.73	2.83	2.83	2.83
	3	3	3	3	3	3	3
	140245812	140248968	140248968	140248968	140248968	140248968	140248968
boyer	4.34	3.28	3.27	3.25	3.52	3.54	3.65
	5	5	5	5	5	5	5
	133717672	133717152	133717152	133717152	133717152	133717152	133717152
mqueens	40.46	23.59	18.84	19.81	23.24	18.74	19.25
	41	45	41	41	45	41	41
	1513225540	1517356516	1513223616	1513223576	1517356516	1513223628	1513223628

5.1 Space Performance of the Collectors

Before discussing the figures in Table 3 we must explain the following: when two garbage collection algorithms running in the same allocator with the same policy can in principle reduce memory requirements to a different extent, there is a first garbage collection that shows a difference in the number of bytes collected and that one reveals the direction of the basic difference. But from that moment on, the future garbage collections, will (or can) be triggered at a *different* moment, and it depends completely on the characteristics of the benchmark whether this benefits one or the other collector. This means that the total number of bytes collected during the total run of a benchmark is an approximation of the characteristics of the algorithm. It is nevertheless meaningful, as programs are usually meant to run until completion and not just until the first garbage collection. But it explains why sometimes a garbage collection algorithm that in principle should perform worse (in the total number of collected bytes) actually performs better. This effect is visible in the tables and typically the differences are small meaning that (as far as the benchmarks go) the difference between the basic algorithms must be considered small. So we will concentrate on the similarities in the table and on the large differences.

- most often optimistic and cautious perform exactly the same, or they differ marginally; Cheney without marking shows even no difference at all between the two,
- dangerous copying performs often the same as the other variants, but for chess and mqueens there are significant differences: in chess, optimistic copying recycles about 30% more garbage; in mqueens, dangerous copying needs 10% more garbage collections,
- mark&Cheney never recycles significantly more garbage than either of the optimistic variants.

This leads to the conclusion that indeed some benchmarks show the problem of multiple copying, but that the optimistic and cautious variants approximate the ideal of mark&copy very closely. Consequently time performance will be decisive in choosing the preferred system.

The differences between recursive copying and Cheney-based copying can be attributed to the fact that these algorithms visit nodes of a term in a different order; the former is depth-first and the latter more breadth-first. This leads in theory to a different collection behaviour and it shows also in practice, but only to a very minor extent.

5.2 Time Performance of the Collectors

The timings for the tsp benchmark are too small to be meaningful, but its space performance seems in line with the other benchmarks. Only for the emul benchmark are the non-marking variants slower than the mark&Cheney. The difference is however very small and one must take into account that for emul there are 75 garbage collections, which in total take less than two seconds. Perhaps emul should not be taken into account and also dnamatch seems rather small although

its timings show the same trend as the rest. Indeed, for all other benchmarks, the non-marking variants outperform mark&Cheney clearly. The following are also worth nothing:

- cautious copying does not offer a performance advantage over optimistic copying: either the difference is very small, or significantly in favour of optimistic copying,
- in most cases dangerous copying is as good as the other variants; but when the benchmark exhibits the problem of multiple copying, dangerous copying loses badly; this is most prominent in the mqueens benchmark,
- Cheney without marking is on average slightly slower than the recursive variant.

As a conclusion we can say that the variants without marking perform significantly better than the original mark&Cheney. This is true for the benchmark with a large root set (chess) as well as for mqueens whose root set is tiny.

Since there is no real space advantage of cautious copying over optimistic copying, the collectors based on optimistic copying come out as winners. Also recursive copying beats Cheney copying narrowly.

6 Cache Behaviour of the Collectors

In Table 4 we show the results obtained by running the various collectors inside a cache simulator. For this purpose we used *cacheprof*, an open-source cache profiler, written by Julian Seward and available from http://www.cacheprof.org. The system traps all memory references during the execution of a benchmark and feeds them to its cache simulator. Afterwards summaries are printed for each function and the corresponding source code is annotated. Simple processing of these data results in the number of cache misses while the collectors were running: the profiler also produces data on the number of memory references and the number of instructions executed, but the performance of the collectors seems most of all determined by the cache misses. The cache misses of the mutator are not shown: they seem hardly influenced by the choice of the collector and this is in line with the observed mutator run times. The cache modelled for these tests was the L2 cache of a Pentium III with the following properties: 256 kb in size, 8 way associative, 32 bytes per line. This is the kind of machine that was also used to obtain the timings in Table 2.

Cache simulation results can usually not explain completely the performance measurements. However, we found a quite significant correlation between the number of cache misses during the execution of the collectors and their run time. Apart from this general observation, we learn from the tables the following:

- the non-marking collectors show very similar results and dangerous copying generates more cache misses than optimistic and cautious copying; this is due to multiple copying and it makes a large difference for chess and mqueens; for these benchmarks the improvement in cache misses for optimistic and cautious copying over dangerous copying results in the noticeable performance improvement in Table 1,
- Cheney without marking generates sometimes more and sometimes less cache misses than recursive copying; the timings also go both ways, but favour recursive copying slightly,

Table 4. Relating the cache behaviour to performance: # of cache misses during garbage collection, total runtime of the collector

	mark&Cheney	recursive copy without marking			Cheney without marking		
benchmark	hProlog	dangerous	optimistic	cautious	dangerous	optimistic	cautious
emul	12191604	11919013	11917323	11917829	11867199	11867386	11867308
	1.83	1.88	1.89	1.91	1.90	1.90	1.92
tsp	184777	142680	142559	142501	141781	141635	141596
	0.05	0.05	0.00	0.02	0.04	0.04	0.03
dnamatch	724950	513563	513487	513506	513094	513000	513005
	0.13	0.07	0.08	0.10	0.12	0.08	0.09
chess	11197160	6340474	6184802	6179661	6340382	6174725	6174724
	2.88	1.77	1.67	1.74	2.09	1.93	2.02
serial	22178379	14942488	14942463	14942555	15040549	15040597	15040654
	3.89	2.72	2.73	2.73	2.83	2.83	2.83
boyer	24307549	17779588	17780083	17780245	19474448	19474397	19474464
	4.34	3.28	3.27	3.25	3.52	3.54	3.65
mqueens	183332375	77078722	62506274	62506337	77077935	62505496	62505414
	40.46	23.59	18.84	19.81	23.24	18.74	19.25

– the memory behaviour of the non-marking collectors is clearly much better than that of the mark&copy collector; optimistic copying gives improvements in cache misses ranging from 23% up to 58% (ignoring the smallish emul); the improvements in time are proportional.

It is clear from these cache simulations that the non-marking collectors have a better cache behaviour than the marking collector, and that this leads to their better performance.

7 Concluding Remarks

One issue has been left open: how to recover from an overflow of the to-space during a garbage collection? Basically, the answer is that the to-space does not need to be contiguous: when it overflows, an extra chunk of to-space can be allocated and later glued to the other chunks at almost no cost. We have not implemented the recovery from to-space overflow, but of course all our variants check for such overflow.

The relation between copying terms and garbage collection is quite old: findall/3 (which uses a form of copying, even if based on assert/1 or record/2) has sometimes been advocated as *the poor man's garbage collection* , because often true garbage collection can be avoided using findall/3. And of course findall/3 has no need for marking. At least three garbage collectors are based on copying without prior marking: the one in [17] uses copying for the most recent segment only. Cells in the most recent segment are known not to be trailed, so reference chains can be shortened and the problem of multiple copying does not exist. Further, both [13] and [19] report on an implementation which avoids the multiple copying of the same cell by postponing the copy of cells that look dangerous. They both report that the set of postponed cells is small in their benchmarks. However, [13] collects only small portions of the heap and [19] collects only the

most recent segment. Neither of them have tried this postponing schema for major collections. On top of that, since hProlog allocates all variables on the heap, more root cells will look dangerous. This makes postponing less attractive. The abstract machine used in [13] is quite different from the WAM, and the garbage collection policies in [13] and [19] make a head on comparison with our collectors very difficult.

We have investigated using two different copying algorithms for garbage collection of the WAM heap, without prior marking. Both were implemented in three variants reflecting different degrees of danger that the heap might grow during a garbage collection. We get a nice speedup of the garbage collection process itself on several benchmarks. Even for a system like hProlog in which many cells are potentially dangerous, the observed memory behaviour is very good, so *a priori* we expect that in systems that follow WAM more closely in their allocation of free variables, copying without prior marking will perform better than with marking and without a big risk to overflow the heap. However, our results do not imply that postponing is viable for major collections, as postponing does not deal in a fundamental way with multiple copying: the cautious schema however does.

Acknowledgements

We thank the (main) authors/implementors of Prolog systems that come with sources: Mats Carlsson (SICStus Prolog) and Vitor S. Costa (Yap) and their teams. We also thank Henk Vandecasteele for his work on the ilProlog compiler used within hProlog.

References

1. H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990 See also: http://www.isg.sfu.ca/~hak/documents/wam.html. 194
2. K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988. 194, 196, 202
3. Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In Y. Bekkers and J. Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 82–102. Springer-Verlag, Sept. 1992. 194
4. J. Bevemyr and T. Lindgren. A simple and efficient copying garbage collector for Prolog. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 88–101. Springer-Verlag, Sept. 1994. 195, 196, 198, 199
5. M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology (KTH), Stokholm, Sweden, Mar. 1990 See also: http://www.sics.se/isl/sicstus.html. 194

6. L. F. Castro and V. S. Costa. Understanding Memory Management in Prolog Systems. In P. Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming, ICLP'2001*, number 2237 in Lecture Notes in Computer Science, pages 11–26. Springer-Verlag, nov 2001. 196, 197, 203
7. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970. 195, 196
8. V. S. Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of PPDP'99*, volume 1702 of *LNCS*, pages 261–277. Springer-Verlag, Sept. 1999 See also http://www.ncc.up.pt/~vsc/Yap/. 194
9. B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999. 194
10. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000. 194
11. B. Demoen, P.-L. Nguyen, and R. Vandeginste. Copy_term/2 and garbage collection. Report CW 329, Dept. of Comp. Sc., K. U.Leuven, Belgium, Jan. 2002. 195
12. R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management*. John Wiley, 1996 See also http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html. 194
13. X. Li. Efficient memory management in a merged heap/stack Prolog machine. In *Proceedings of the 2nd ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 245–256. ACM Press, 2000. 197, 199, 202, 206, 207
14. F. L. Morris. A time- and space-efficient garbage compaction algorithm. *Communications of the ACM*, 21(8):662–665, Aug. 1978. 195, 196
15. D. Sahlin and M. Carlsson. Variable Shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991. 195
16. P. Tarau and U. Neumerkel. A novel term compression scheme and data representation in the binwam. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 73–87. Springer-Verlag, Sept. 1994. 200
17. H. Touati and T. Hama. A light-weight Prolog garbage collector. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 922–930. OHMSHA Ltd. Tokyo and Springer-Verlag, nov/dec 1988. 206
18. D. H. D. Warren. An abstract Prolog instruction set. Tech. Report 309, SRI, 1983. 194
19. N.-F. Zhou. Garbage Collection in B-Prolog. In B. Demoen, editor, *Proceedings of the First Workshop on Memory Management in Logic Programming Implementations, co-located with CL2000*, pages 1–10, http://www.cs.kuleuven.ac.be/~bmd/mmws.html, July 2000. 199, 202, 206, 207

Logical Algorithms

Harald Ganzinger[1] and David McAllester[2]

[1] MPI Informatik
D-66123 Saarbrücken, Germany
hg@mpi-sb.mpg.de
[2] AT&T Labs-Research

Abstract. It is widely accepted that many algorithms can be concisely and clearly expressed as logical inference rules. However, logic programming has been inappropriate for the study of the running time of algorithms because there has not been a clear and precise model of the run time of a logic program. We present a logic programming model of computation appropriate for the study of the run time of a wide variety of algorithms.

1 Introduction

It is widely accepted that many algorithms can be concisely and clearly expressed as logical inference rules. Unfortunately there is a fundamental problem in using inference rules in the study of algorithms — the precise running time of inference rules can depend on implementation details such as indexing. Run time is one of the most fundamental properties of any algorithm. Textbooks on algorithms and data structures begin by giving a model of computation — usually a random access memory (RAM) machine — which formally defines the notion of run time. Algorithms must be presented in a way that makes their execution on the underlying model sufficiently clear that precise run time analysis is possible. We will use the phrase *algorithmic* model of computation to mean a model of computation that includes a well defined notion of run time. Here we present a new algorithmic logic programming model of computation. We prove that the abstract notion of run time in this model is implementable on a RAM machine extended with constant time hash table operations. We also show how a variety of algorithms that are not easily expressed in earlier algorithmic logic programming models are naturally expressed in our new model.

McAllester recently introduced a pure logic programming algorithmic model. In this model a program is simply a finite set of first order Horn clauses [5]. The program (set of rules) R defines a mapping from an input database D to an output database $R(D)$ where $R(D)$ is all assertions derivable from the assertions in D using the rules in R. The running time of R on input D is defined to be the number of assertions in $R(D)$ plus the number of "prefix firings" in the computation. A prefix firing is an instance of a prefix of the antecedents of a rule such that the prefix instance is true in the final closure $R(D)$ (see [5] for details). McAllester proved that an interpreter can be constructed for pure logic

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 209–223, 2002.
© Springer-Verlag Berlin Heidelberg 2002

programs such that the program can be executed on a RAM machine extended with constant time hash table operations in time proportional to its abstract run time.

A variety of efficient algorithms can be naturally expressed as pure logic programs, especially dynamic programming algorithms in parsing and in the static analysis of computer programs. However, many other problems appear not to have efficient solutions as pure programs. For example, pure logic programs do not seem to support an efficient union-find algorithm. In the same paper, McAllester also presents a more elaborate logic-programming model that incorporates union-find specifically into the model. More recently Ganzinger and McAllester [2] gave a more general algorithmic logic programming model, one including rule priorities and deletion, which allows union-find to be implemented efficiently, as well as many other algorithms not naturally covered by the pure logic programming model.

While rule priorities and deletions greatly expand the power of the model, they still fail to cover certain algorithms. Dijkstra shortest path, for example, still appears to not have a natural representation. The contribution of this paper is an algorithmic logic programming model with yet broader coverage. In particular we allow different instances of the same rule to run at different priorities. We prove that an interpreter (or compiler) can be constructed on a RAM machine extended with constant time hash table operations such that the time taken to run a program is proportional to the abstract notion of run time. As evidence for the power of our model we demonstrate implementations of Dijkstra's shortest path and of minimal spanning tree.

If one ignores the abstract notion of run time, our logic programming model is not particularly novel. The language is variant of bottom-up logic programming. Bottom-up logic programming has been widely studied in the context of deductive databases [13,11,6,12]. Bottom-up logic programming is closely related to "memoing" or "tabling" for Prolog programs [10,9,1]. The bottom-up language described here allows deletion. Our notion of deletion is superficially similar to widely studied notions of negation in logic programming such as well founded (stratified) programs [8,4] and stable-model semantics [3]. Here, however, we use a "don't-care" nondeterministic semantics which does not require the program to be well founded and where the final database need not be stable. Deletion in logic programming has also been modeled with linear logic [7]. The linear logic approach is similar to ours but it has not been given an abstract notion of run time executable on a RAM machine. Our notion of rule priorities is similar to notions previously studied in the logic programming literature, e.g., [14]. But to our knowledge earlier work has not provided an abstract notion of run time implementable on a RAM machine.

2 A Broad Algorithmic Logic Programming Model

To support a wide variety of algorithms it is necessary to support arithmetic. To support arithmetic we define atomic formulas and terms with the following grammar.

$$\begin{aligned}
&\Phi \equiv P(\tau_1,\ \ldots,\ \tau_k) \ \mid\ N_1 < N_2 \ \mid\ N_1 \leq N_2\\
&\tau \equiv H \ \mid\ N\\
&H \equiv x \ \mid\ f(\tau_1,\ \ldots,\ \tau_k)\\
&N \equiv i \ \mid\ n \ \mid\ N_1 + N_2 \ \mid\ N_1 * N_2
\end{aligned}$$

This is two-sorted grammar with a sort for Herbrand terms (H) and a separate sort for integers (N). We allow predicates and functions to take either sort as arguments although we assume that each predicate and function symbol has a specified arity and a specified sort for each argument. In the expression $P(\tau_1,\ \ldots,\ \tau_k)$ we require that P be a predicate symbol of arity (number of arguments) k and that each argument is of the appropriate sort for the predicate P. A similar comment applies to the expression $f(\tau_1,\ \ldots,\ \tau_k)$ in the grammar for H. The function symbol f in the grammar for Herbrand terms should be viewed as a data constructor. The precise semantics is defined below. Note that there are two sorts of variables — Herbrand variables such as x in the grammar for Herbrand terms and integer variables such as i in the grammar for integer terms. In the grammar for integer terms, n ranges over any integer constant.

A *ground term* is a term not containing variables or arithmetic operations. A numerical constant such as 5 is a ground term. A constant symbol such as c is also a ground term where we view c as a function of no arguments. The term $f(c, 4)$ is also a valid ground term provided that f is a function of two arguments, the first of sort Herbrand term and the second of sort integer. The term $f(c, 2+3)$ is not a valid ground term because it contains an arithmetic operation but the term $f(c, 5)$ is. A *ground substitution* is a mapping from variables to ground terms where the value of a Herbrand variable must be a ground term of the form $f(\tau_1,\ \cdots,\ \tau_k)$ and the value of an integer variable must be an integer constant. If τ is a term (as defined by the above grammar) and σ is a ground substitution, then we define $\sigma(\tau)$ to be the result of replacing each variable by its value in σ and then computing the result of all arithmetic operations. For example, if σ maps x to c and i to 3, then $\sigma(f(x,\ i + 4))$ is the ground term $f(c, 7)$. A ground atom is either one of the constants T or F (representing "true" and " false" respectively) or an expression of the form $P(\tau_1,\ \ldots,\ \tau_k)$ where each τ_i is a ground term. If Φ is an atomic formula as defined by the above grammar, and σ is a ground substitution, then we define $\sigma(\Phi)$ to be the result of replacing each variable by its value and then computing arithmetic operations. For example, if σ maps i to 5 then $\sigma(P(c, i))$ is the ground atom $P(c, 5)$ while $\sigma(i < 7)$ is the constant T and $\sigma(7 < i)$ is the constant F.

Here we are interested in inference rules (logic programs) that support deletion. An expression of the form del(A) where A is an atomic formula (as defined by the above grammar) will be called a *deletion assertion*. We use del(A) rather

than $\neg A$ to avoid any confusion with the semantic notion of logical negation. We will use the term *assertion* to mean either a ground atom or a deletion assertion. A deletion assertion $\mathrm{del}(A)$ is called *ground* if A is a ground atom. We define an *inference rule* to be an expression of the form $A_1 \wedge \ldots \wedge A_n \Rightarrow C$ where each *antecedent* A_i is an assertion (atoms or deletion assertions) not involving addition or multiplication and the *conclusion* C is a finite set of assertions (atoms or deletion assertions) such that every variable in C occurs in some antecedent and all variables in a comparison antecedent, i.e., an antecedent involving $<$ or $\leq$, must occur in some earlier antecedent. A *state* of the computation is a finite set of ground assertions. In the computational processes defined here, states grow monotonically over time — the deletion of A is modeled by the addition of $\mathrm{del}(A)$. An atom A is said to be *visible* in state S if A is the constant T or A is a ground atom such that $A \in S$ and $\mathrm{del}(A) \not\in S$, or A is a deletion assertion and $A \in S$.

Deletion makes the behavior of rule sets nondeterministic. For example, consider the following rules with deletion.

$$p \Rightarrow q \qquad q \Rightarrow s, \mathrm{del}(q) \qquad q \Rightarrow w, \mathrm{del}(q)$$

Suppose the initial database contains only p. The first rule fires adding the assertion q. Now either the second or third rule can fire, but once one of them fires the other is blocked because q is no longer visible. Hence the final state is nondeterministically either $\{p,\ q,\ \mathrm{del}(q),\ s\}$ or $\{p,\ q,\ \mathrm{del}(q),\ w\}$. When viewing rules with deletions as algorithms, this nondeterminism is viewed as "don't care" nondeterminism — choices between equal priority rule invocations are made arbitrarily and are not backtracked (priorities are discussed below). As another example consider the following rules.

$$p \Rightarrow q \qquad q \Rightarrow \mathrm{del}(q), w \qquad w \Rightarrow \mathrm{del}(w), q$$

A naive interpretation of deletion might lead one to think that starting from the state $\{p\}$ we get an infinite loop switching between a state where q is visible and a state where w is visible. But under the semantics used here, once an atom is deleted it remains invisible forever. Running the above rules from the start state $\{p\}$ terminates with the state $\{p,\ q,\ \mathrm{del}(q),\ w,\ \mathrm{del}(w)\}$. Our notion of deletion is quite different from semantic negation.

Priorities allow fine grain control for rules with deletion. We define a *program* to be a pair $\langle R,\ \pi\rangle$ where R is a finite set of rules (as defined above) and π maps each rule to an arithmetic expression constructed from integer variables occurring in the first antecedent of the rule. We use the convention that smaller integers represent higher priorities. For $r \in R$, and σ a ground substitution interpreting at least the variables of the first antecedent of r, we let $\pi(r,\ \sigma)$ be the maximum of 1 and the integer $\sigma(\pi(r))$. Note that "priority 1" is the highest possible priority. We say that r has *fixed priority* if $\pi(r)$ is an integer constant, and otherwise we say that r has *variable priority*.

Now consider a fixed program $\langle R,\ \pi\rangle$. An *instance* of r is a pair $\langle r,\ \sigma\rangle$ where σ is a ground substitution defined on (only) the variables occurring in the antecedents of r (and hence also defined on the variables in the conclusion). We

will say that the antecedents of the instance $\langle r,\ \sigma\rangle$ *hold* in a state S if, for each antecedent A_i of r, we have that $\sigma(A_i)$ holds in S. We say that an instance $\langle r,\ \sigma\rangle$ is *pending* at state S if $r \in R$; the antecedents of the instance hold in S; and $S \not\models S \cup \sigma(C)$ where C is the conclusion of r. We define the *priority of a state* S, written $\pi(S)$, to be the priority of the highest priority pending rule instance, i.e. $\pi(r,\ \sigma)$ for some pending instance $\langle r,\ \sigma\rangle$ such that there is no pending instance $\langle r',\ \sigma'\rangle$ with $\pi(r',\ \sigma') < \pi(r,\ \sigma)$. If there is no pending rule instance for S then S is called *saturated* and we define $\pi(S)$ to be (positive) infinity. We write $S \overset{R,\pi}{\rightarrow} S'$ if there exists some instance $\langle r,\ \sigma\rangle$ pending in S with $\pi(r,\sigma) = \pi(S)$ and where $S' = S \cup \sigma(C)$ where C is the conclusion of r. An $\langle R, \pi\rangle$-computation is a sequence $S_0,\ S_1,\ \ldots,\ S_t$ where $S_i \overset{R,\pi}{\rightarrow} S_{i+1}$. We also allow infinite computations of the form $S_0,\ S_1,\ S_2,\ \ldots$. A program $\langle R, \pi\rangle$ is said to terminate provided that there are no infinite $\langle R, \pi\rangle$-computations. An $\langle R, \pi\rangle$-computation is called *complete* if it is finite and the final state is saturated. If there exists a complete $\langle R,\ \pi\rangle$-computation from S_0 to S_t then S_t is called an $\langle R,\ \pi\rangle$-*saturation* of S_0.

We now define an abstract notion of running time for terminating programs. Consider a rule $r \in R$. If r has n antecedents then for $1 \leq j \leq n$ we let r_j denote the jth antecedent of r. A *prefix instance* of r is a triple $\langle r,\ i,\ \sigma\rangle$ where $1 \leq i \leq n$ and σ is a ground substitution defined on (only) the variables occurring in the first i antecedents $r_1,\ \ldots,\ r_i$. An instance of r as defined earlier is just a prefix instance with $i = n$.

> **Definition of Abstract Running Time:** Consider a complete finite computation C starting in state S_0 and ending in state S_t. A *weak prefix firing* of r in C is a prefix instance $\langle r,\ i,\ \sigma\rangle$ of r such that for $1 \leq j \leq i$ we have that either $\sigma(r_j)$ is T or $\sigma(r_j) \in S_t$ (note that elements of S_t need not hold in S_t — S_t may contain both A and del(A)). A *strong prefix firing* of r is a weak prefix firing $\langle r,\ i,\ \sigma\rangle$ such that there exists a state S with $\pi(S) \geq \pi(r,\ \sigma)$ such that all the antecedents of the firing hold in S, i.e., for all $1 \leq j \leq i$ we have that $\sigma(r_j)$ holds in S. An *antecedent instance* of a rule r is an element of S_t that is an instance of an antecedent of r. The abstract running time of the computation is defined to be $|S_0| + P_f + (P_v + A_v)\log N$ where $|S_0|$ is the number of assertions in S_0, P_f is the number of strong prefix firings of fixed priority rules; P_v is the number of strong prefix firings of variable priority rules; A_v is the number of antecedent instances of variable priority rules; and N is the number of distinct priorities, i.e., the number of priorities of the form $\pi(r, \sigma)$ with $\sigma(r_1) \in S_t$.

Without deletion there is no distinction between week and strong prefix firings — any weak prefix firing is strong by virtue of the final state S_t which as infinite priority. Also note that an assertion which is asserted and then deleted, both at priority higher than the priority of a rule r, does not participate in strong prefix firings of r. However, if such an assertion matches an antecedent of r then it counts as an antecedent instance no matter how quickly it is deleted. As a simple case consider the rule $P(x, y) \wedge P(y, z) \wedge Q(y) \Rightarrow P(x, z)$ with fixed

$$\texttt{(B1,1)}\ \frac{E(v,u)}{E(u,v)} \qquad \texttt{(B2,1)}\ \frac{E(v,u)}{\mathsf{unlabeled}(u)} \qquad \texttt{(B3,1)}\ \frac{\mathsf{labeled}(u,k)}{\mathrm{del}(\mathsf{unlabeled}(u))}$$

$$\texttt{(B4,1)}\ \frac{\mathsf{labeled}(u,A) \quad E(u,v)}{\mathsf{labeled}(v,B)} \qquad \texttt{(B5,1)}\ \frac{\mathsf{labeled}(u,B) \quad E(u,v)}{\mathsf{labeled}(v,A)} \qquad \texttt{(B6,2)}\ \frac{\mathsf{unlabeled}(u)}{\mathsf{labeled}(u,A)}$$

Fig. 1. Checking for Bipartiteness

priority 1 and assume there is no deletion. Any firing of the first two antecedents corresponds to at most one firing of all three antecedents but many firings of the first two antecedents may fail to correspond to any firing of all three. Note that the (logically equivalent) rule $P(x,y) \wedge Q(y) \wedge P(y,z) \Rightarrow P(x,z)$ is more efficient, i.e., has fewer prefix firings. We rely on the programmer to write rules in an efficient form given an understanding of the abstract notion of run time. The main result of this paper is the following which extends the result in [2] beyond fixed priority rules.

Theorem 1. *For any terminating program $\langle R,\ \pi\rangle$ there exists an algorithm running on a RAM machine extended with constant time hash table operations such that for any given initial state S_0 the algorithm computes a complete $\langle R,\ \pi\rangle$-computation from S_0 in time proportional to the abstract running time of the generated computation.*

3 Examples

Our first example is a simple algorithm for determining whether a given input graph is bipartite. A graph is bipartite if its nodes can be partitioned into two subsets A and B such that edges do not connect any pair of nodes of the same subset. The rules in figure 1 determine bipartiteness — the graph is bipartite unless these rules assign both labels to the same node. Each rule is labeled with a name and a priority. In this example all rules have fixed priority. Furthermore, all rules have priority 1 except rule (B6) which has (lower) priority 2. Note that in any state with priority 2 or greater we have that $\mathsf{unlabeled}(u)$ is visible if and only if there is no assertion of the form $\mathsf{labeled}(u,\ k)$. Since all rules in the example have fixed priority, the abstract running time is just the size of the input data base plus the number of strong prefix firings of the rules. One can check that every rule has at most $O(e)$ (weak) prefix firings where e is the number of edges in the input graph. Hence the abstract running time of the algorithm is linear in the size of the input graph.

The Dijkstra shortest path algorithm is depicted in Figure 2 where the assertion $E(u, c, v)$ represents a directed edge from u to v with integer distance c. Let e be the number of input edges. We assume all distances are non-negative and that every source node is contained in at least one edge so that the number of nodes is at most twice the number of edges. The rule set derives assertions of the form $\mathsf{dist}(v, d)$ stating that the shortest path from the given source node to v is no longer than d. Note that rule (D2) ensures that in any state with priority 2 or greater we have at most one bound associated with each node. It is easy to see that when the computation terminates each bound is equal to the actual minimal distance — the procedure is correct. Furthermore, for any state transition cause by an instance of (D3) on node v and distance d we have that all future state transitions involve distances at least as large as d and so d is the final distance, i.e., the true shortest distance to d. This implies that all distance assertions are either of the form $\mathsf{dist}(v, 0)$, where v is the source node, or of the form $\mathsf{dist}(u, d+c)$ such that some edge ending in u with cost c starts a node with minimal distance d. This implies that there are at most $O(e)$ distance assertions ever asserted and that the final state (including all deleted assertions) is of size $O(e)$. Now consider the abstract running time of this procedure. When rule (D3) derives a new distance, rule (D2) immediately deletes the larger distance. In a strong prefix firing all antecedents must be simultaneously true at the same state. This implies there is one strong prefix firing of (D2) for each state transition induced by rule (D3). Each state transitions caused by rule (D3) corresponds to a strong prefix firing of (D3) so the number of such transitions can be no larger than the number of strong prefix firings of (D3). So the number of strong prefix firings of (D2) is bounded by the number of strong prefix firings of (D3). To bound the number of strong prefix firings of (D3) we show that in any strong prefix firings of (D3) we must have that d is the unique shortest distance to the node v. To see this consider any strong prefix firing of (D3) involving v and d. By definition there must be a state S with $\pi(S) \geq d+2$ such that $\mathsf{dist}(v, d)$ holds in S. But for this rule set, if $\pi(S) \geq d+2$ then the next state transition must use an instance of (D3) involving a distance d or larger. This implies that all bounds derived after this point will also be of distance d or larger. So d must be the final distance bound for v, i.e., the true shortest distance to v. So in strong prefix firings of (D3) each node v is always associated with the same distance d. This implies that the number of strong prefix firings of (D3) is $O(e)$. So we have that $|S_0|$ is $O(e)$, P_f is $O(e)$, P_v is $O(e)$ and A_v (the number of antecedent instances of variable priority rules) is no larger than $|S_t|$ which is $O(e)$. So the abstract complexity is $O(e \log e)$.

Figure 3 shows a logic programming implementation of union-find. The rules maintain the invariant that the find relation is functional — for any x there is at most one y such that $\mathsf{find}(x, y)$ is visible. Furthermore, any two nodes x and y have the same find value if and only if they are equivalent in the smallest equivalence relation defined by the union operations. Figure 3 is essentially the implementation given in [2] and we do not give a detailed analysis here. This implementation uses greedy path compression. The total abstract running time

$$\texttt{(D1,1)}\ \frac{\mathsf{source}(v)}{\mathsf{dist}(v,0)} \qquad \texttt{(D2,1)}\ \frac{\mathsf{dist}(v,d)\quad \mathsf{dist}(v,d')\quad d'<d}{\mathrm{del}(\mathsf{dist}(v,d))} \qquad \texttt{(D3,}d\texttt{+2)}\ \frac{\mathsf{dist}(v,d)\quad E(v,c,u)}{\mathsf{dist}(u,d+c)}$$

Fig. 2. Dijkstra Shortest Path

of all the union-find rules in figure 3 is $O(n \log n)$ where n is the total number of union operations in the computation, i.e., the total number of union assertions, including deleted assertions, in the final state. Furthermore, for any fixed node x, the number of nodes y such that $\mathsf{find}(x, y)$ is ever asserted is at most $\log n$ (every time the find of a node changes the size of that node's equivalence class at least doubles). It is possible to implement a lazy path compression version of union-find that has inverse-Ackermann running time but the implementation is somewhat more involved and the interface to the rules is more complex[1]

Figure 4 gives an algorithm for computing a minimum spanning tree of a connected undirected graph. These rules take a set of input arcs of the form $E(x, c, y)$ which states that there exists an undirected arc between x and y with cost c. The rules produce output arcs of the form $\mathsf{out}(x, c, y)$. The rules use the union-find module given in figure 3 to maintain an equivalence relation on nodes where x and y are equivalent if and only if there exists a path in the *output* arcs from x to y. The rules also maintain the invariant that the set of output edges form a subset of some minimum spanning tree. To see this consider an invocation of rule (ST2) with first antecedent $E(x, c, y)$. The edge in the first antecedent connects two unequal nodes and is minimum cost over all edges connecting nonequivalent nodes. By the induction hypothesis there exists a minimum spanning tree containing all existing output edges. Let S be such a minimum spanning tree. If S contains the edge $E(x, c, y)$ then this invocation of the rule preserves the invariant. If S does not contain $E(x, c, y)$ then consider the path in S from the equivalence class of x to the equivalence of y under the equivalence relation defined by the current output edges where each current equivalence class is viewed as a single node. Remove any edge in this path from S. The resulting graph has two components — one containing x and one containing y. Now add the edge $E(x, c, y)$. The result is a spanning tree of cost no larger than the original. This new spanning tree is also minimum. Hence there exists a minimum spanning tree containing $E(x, c, y)$ and the invariant is maintained. The output edges of the final state must then be a minimum spanning tree of the entire input graph.

Now consider the abstract running time of the algorithm shown in figure 4. Let n be the number of nodes and e be the number of edges in the input graph.

[1] The rules in figure 3 take union assertions as input and produce find assertions as output. An inverse-Ackermann version requires on-demand find requests as additional input.

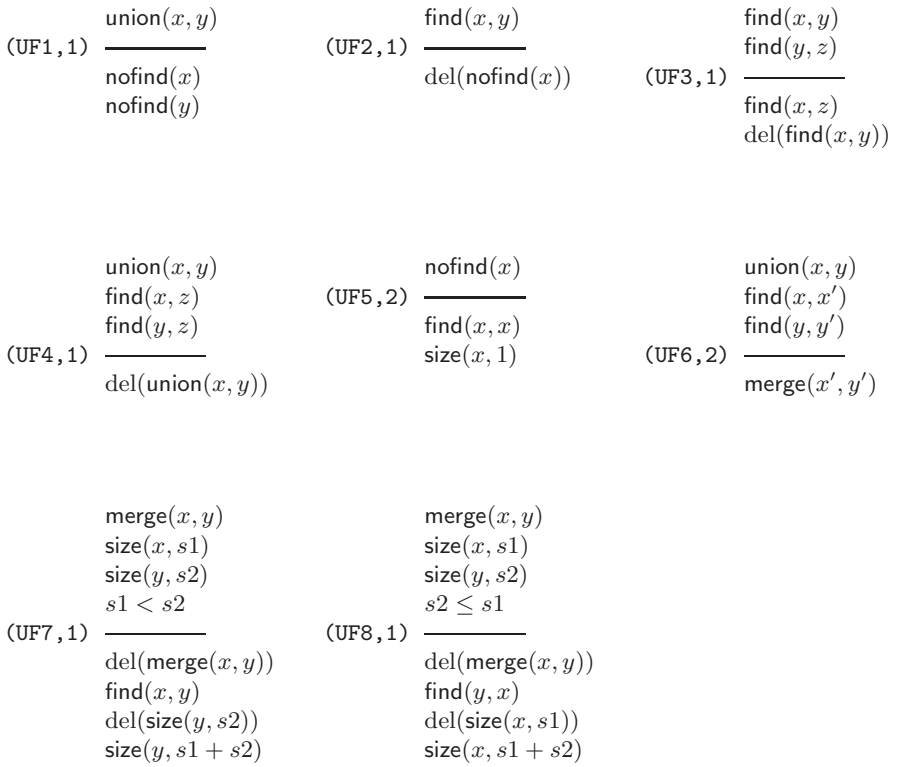


Fig. 3. Union-Find

The rules can generate at most $n-1$ union assertions before all nodes become equivalent. Since there are at most $n-1$ union assertions, the abstract running time of the union-find rules is $O(n \log n)$. In any state of priority 2 or greater there is at most one find value for each node. Furthermore, for a given x there can be at most $\log n$ values y such that $\mathsf{find}(x, y)$ is ever asserted. This implies that there can be at most $O(e \log n)$ strong prefix firings of rule (ST1). Since this rule has a fixed priority, its abstract running time is $O(e \log n)$. Finally, there are at most e prefix firings and at most e antecedent instances of rule (ST2) and at most e different priorities associated with this rule. Hence the abstract running time of rule (ST2) is no larger than $O(e \log e)$. So we get a total abstract running time of $O(e \log e)$.

4 Proof of Theorem 1

We first use a series of run-time preserving rule transformations to put rules in a standard form. First we show that it suffices to consider rules with at most two antecedents and with no more than one assertion in the conclusion. We

$$\begin{array}{rl} & E(x,c,y) \\ & \mathsf{find}(x,z) \\ & \mathsf{find}(y,z) \\ \texttt{(ST1,3)} & \hline \\ & \mathrm{del}(E(x,c,y)) \end{array} \qquad\qquad \begin{array}{rl} & E(x,c,y) \\ \texttt{(ST2,c+4)} & \hline \\ & \mathsf{union}(x,y) \\ & \mathsf{out}(x,c,y) \end{array}$$

Fig. 4. Minimum Spanning Tree

start by replacing the given priority assignment π by the assignment π' where $\pi'(r) = 2 * \pi(r) + 2$. In the resulting program every priority is even and no rule has priority 2. Now consider a rule r of the form $A_1 \wedge \cdots \wedge A_n \Rightarrow \{\Phi_1, \ldots, \Phi_m\}$. We replace r by the following where P_1, P_2, ..., P_n, and Q are fresh predicate symbols and $x_1, \ldots, x_{k_i}$ are the variables occurring in the first i antecedents. The predicate P_i represents the relation defined by the first i antecedents, and $\mathrm{del}(P_i(x_1, \ldots, x_{k_i}))$ represents the statement that some antecedent has been deleted.

$$\begin{aligned} A_1 &\Rightarrow P_1(x_1, \ldots, x_{k_1}) \\ \mathrm{del}(A_1) &\Rightarrow \mathrm{del}(P_1(x_1, \ldots, x_{k_1})) \\ P_1(x_1, \ldots, x_{k_1}) \wedge A_2 &\Rightarrow P_2(x_1, \ldots, x_{k_2}) \\ P_2(x_1, \ldots, x_{k_2}) \wedge \mathrm{del}(P_1(x_1, \ldots, x_{k_1})) &\Rightarrow \mathrm{del}(P_2(x_1, \ldots, x_{k_2})) \\ P_2(x_1, \ldots, x_{k_2}) \wedge \mathrm{del}(A_2) &\Rightarrow \mathrm{del}(P_2(x_1, \ldots, x_{k_2})) \\ &\vdots \\ P_{n-1}(x_1, \ldots, x_{k_1}) \wedge A_n &\Rightarrow P_n(x_1, \ldots, x_{k_2}) \\ P_n(x_1, \ldots, x_{k_2}) \wedge \mathrm{del}(P_{n-1}(x_1, \ldots, x_{k_1})) &\Rightarrow \mathrm{del}(P_n(x_1, \ldots, x_{k_2})) \\ P_n(x_1, \ldots, x_{k_2}) \wedge \mathrm{del}(A_n) &\Rightarrow \mathrm{del}(P_n(x_1, \ldots, x_{k_2})) \\ P_n(x_1, \ldots, x_{k_n}) &\Rightarrow Q(x_1, \ldots, x_{k_n}) \\ Q(x_1, \ldots, x_{k_n}) &\Rightarrow \Phi_1 \\ &\vdots \\ Q(x_1, \ldots, x_{k_n}) &\Rightarrow \Phi_m \end{aligned}$$

If some antecedent of the original rule r is of the form $\mathrm{del}(B_j)$ then the above rules include rules with double deleted antecedents, i.e., rules with antecedents of the form $\mathrm{del}(\mathrm{del}(B_j))$. Rules with double deletion antecedents are simply thrown out — they are not included in the result of the transformation (deletion assertions can not be deleted). Rules containing deletions of arithmetic comparisons, e.g., $\mathrm{del}(i < j)$ are also ignored. All rules with conclusions of the form $\mathrm{del}(P_i(x_1, \ldots, x_{k_i}))$ or Φ_j have priority 1; rules with conclusions of the form $P_i(x_1, \ldots, x_{k_i})$ have priority $\pi(r) - 1$, and the rule $P_n(x_1, \ldots, x_{k_n}) \Rightarrow Q(x_1, \ldots, x_{k_n})$ has priority $\pi(r)$. In a computation under the transformed program, states of the original computation correspond even priority states of the transformed program. Odd priority states of the transformed program are used to compute the next even-priority state. We leave it to the reader to verify that

the sequence of even priority states, restricted to the predicates of the original program, yields a valid computation of the original program.

We also have to argue that the abstract running time of the transformed program is no more than a constant factor more than the abstract running time of the original program. We do this by charging each prefix firing of the transformed program to some prefix firing of the original program. Since the deletion rules run at the highest possible priority, if $P_i(x_1, \ldots, x_{k_i})$ ever holds in any state then there exists a prefix firing in the even-state computation corresponding to this assertion. This implies that a firing of a rule whose conclusion is of the form $\mathrm{del}(P_i(x_1, \ldots, x_{k_i}))$ can be charged to the prefix of r corresponding to the assertion being deleted in the conclusion. We also have that if $P_i(x_1, \ldots, x_{k_i})$ holds in a state with priority $\pi(r) - 1$ then the corresponding prefix firing of r must hold in that state. This implies that each firing of a rule whose conclusion is of the form $\mathrm{del}(P_i(x_1, \ldots, x_{k_i}))$ corresponds to a prefix firing of the original rule. Each firing of $P_n(x_1, \ldots, x_{k_n}) \Rightarrow Q(x_1, \ldots, x_{k_n})$ or a rule with a conclusion of the form Φ_j corresponds to a full firing of the original rule. A similar argument can be used to show that the number antecedent instances of variable priority rules of the new program is no more than a constant times the number of antecedent instances of variable priority rules of the old program.

We have now shown that we can assume without loss of generality that each rule contains at most two antecedents and only a single conclusion. We now simplify the rules further. Starting with a set of rules with at most two antecedents and only a single conclusion replace the priority assignment π by π' where $\pi'(r) = \pi(r) + 1$. Consider a rule r with two antecedents $A_1 \wedge A_2 \Rightarrow B$. The definition of an inference rule requires that any variable in a comparison antecedent occurs in some earlier antecedent. This implies that the first antecedent is not a comparison. We first consider the case where the second antecedent is also not a comparison — the case where the second antecedent is a comparison is discussed below. If the second antecedent is not a comparison we replace r by the following set of rules where $x_1, \ldots, x_n$ are all variables occurring in A_1 but not A_2, $y_1, \ldots, y_m$ are all variables that occur in both A_1 and A_2, and $z_1, \ldots, z_k$ are all variables that occur in A_2 but not A_1. The predicates P, and Q, and the function symbols f, g, and h are all fresh.

$$\begin{gathered}
A_1 \Rightarrow P(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)) \\
\mathrm{del}(A_1) \Rightarrow \mathrm{del}(P(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m))) \\
A_2 \Rightarrow Q(g(y_1, \ldots, y_m), h(z_1, \ldots, z_k)) \\
\mathrm{del}(A_2) \Rightarrow \mathrm{del}(Q(g(y_1, \ldots, y_m), h(z_1, \ldots, z_k))) \\
P(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)) \wedge Q(g(y_1, \ldots, y_m), h(z_1, \ldots, z_k)) \Rightarrow B
\end{gathered}$$

The last rule is given the same priority as the original and the other rules are given priority one. The states with priority greater than one correspond to the states of the original rule set. We can charge the firings of the rules with a single antecedent of the form A_1, A_2, $\mathrm{del}(A_1)$, or $\mathrm{del}(A_2)$ to the rules asserting A_1 or A_2.

We can now assume without loss of generality that every rule is either of the form $A \Rightarrow B$ where A is not a comparison, or of the form $A_1 \wedge A_2 \Rightarrow B$

where A_1 is not a comparison, and either A_2 is a comparison with all variables of A_2 occurring in A_1, or else the rule is of the form $P(v, v') \wedge Q(v', v'') \Rightarrow B$ where the three terms v, v', and v'' do not share any variables. These rules will be called *unary rules*, *comparison rules*, and *binary rules* respectively.

We now define a *firing tuple* to be a tuple of the form $\langle r,\ A_1\rangle$ or $\langle r,\ A_1,\ A_2\rangle$ where r is a rule and A_1 is a ground assertion matching the first antecedent of r and A_2 is a ground assertion matching the second antecedent of r. To implement the saturation algorithm we maintain a global priority queue $\mathcal{Q}$ of not-yet-processed firing tuples and a set $\mathcal{R}$ of already-processed firing tuples. The priority of a tuple involving r and A_1 is defined to be $\pi(r,\ \sigma)$ where σ is the substitution resulting from matching the first antecedent of r with A. The algorithm iteratively removes and processes tuples from $\mathcal{Q}$. In addition, for each binary rule r of the form $P(v, v') \wedge Q(v', v'') \Rightarrow B$, and for certain ground terms t we maintain a data structure $\mathcal{W}(r,\ t)$ representing potential full firings of r of the form $P(s, t)$, $Q(t, u)$. More specifically, $\mathcal{W}(r,\ t)$ is an alternating sequence of P-blocks and Q-atoms where a P-block is a possibly empty set of ground P-atoms of the form $P(s, t)$ matching $P(v, v')$ and a Q-atom is an ground assertion of the form $Q(t, u)$ matching $Q(v', v'')$ and any given assertion appears in $\mathcal{W}(r,\ t)$ in at most one place. The sequence $\mathcal{W}(r,\ t)$ starts and ends with (possibly empty) P-blocks. Atoms in the last P-block are called *completed*. The priority of an element $P(s, t)$ of a P-block of $\mathcal{W}(r,\ t)$ is $\pi(r,\ \sigma)$ where σ is the corresponding substitution. Each nonempty P-block is associated with a unique element of highest priority with ties brocken arbitrarily. We will refer to this selected element simply as "the" highest priority element of the P-block. The saturation procedure maintains the following $\mathcal{Q}$ invariant which determines the set of tuples in $\mathcal{Q}$ as a function of the current state S, the set $\mathcal{R}$ of processed firings, and the state of the data structures of the form $\mathcal{W}(r,\ t)$.

> $\mathcal{Q}$-**Invariant:** The queue $\mathcal{Q}$ contains a pair of the form $\langle\ r,\ A\rangle$ if and only if A is visible in the current state; r is a rule in the program whose first antecedent matches A; and the pair $\langle\ r,\ A\rangle$ is not contained on the list $\mathcal{R}$. An triple of the form $\langle\ r,\ P(s, t),\ Q(t, u)\rangle$ is on the queue $\mathcal{Q}$ if and only if $P(s, t)$ and $Q(t, u)$ are both contained in the data structure $\mathcal{W}(r,\ t)$ and $P(s, t)$ is a highest priority element of the P-block immediately preceding the Q-assertion $Q(t, u)$.

Procedure $\mathsf{Assert}(A)$
If A is not visible in S do the following

1. Add A to S.
2. If A is of the form del(B) then remove B and tuples involving B from $\mathcal{Q}$ and all data structures of the form $\mathcal{W}(r,\ t)$. For each Q-assertion removed from a data structure of the form $\mathcal{W}(r,\ t)$ merge the two adjacent P-blocks updating $\mathcal{Q}$ as necessary to maintain the $\mathcal{Q}$-invariant.
3. For each rule r such that A matches the first antecedent of r add the pair $\langle r,\ A\rangle$ to $\mathcal{Q}$.

4. For each binary rule r such that A matches the second antecedent of r add A as a new Q-atom at the end of $\mathcal{W}(r,\ t)$, where t is the first argument of A, followed by a new empty P-block (initialize $\mathcal{W}(r,\ t)$ with an empty P-block if necessary). If the P-block preceding Q is non-empty add the triple $\langle r,\ B,\ A\rangle$ to $\mathcal{Q}$ where B is the highest priority P-assertion in the P-block preceding Q.

Saturation Procedure Let D be a given initial database.

1. Initialize $\mathcal{Q}$ and the state S to be empty.
2. Assert each element of D.
3. While $\mathcal{Q}$ is not empty do the following:
   (i) Remove the highest priority tuple z on $\mathcal{Q}$ and add it to $\mathcal{R}$.
   (ii) If z is $\langle r,\ A\rangle$ for unary r assert the corresponding conclusion of r.
   (iii) If z is $\langle r,\ A\rangle$ for a comparison rule r assert the corresponding conclusion of r provided that the comparison holds.
   (iv) If z is $\langle r,\ P(s,t)\rangle$ where r is a binary rule then add $P(s,t)$ to the first P-block of $\mathcal{W}(r,\ t)$ (initialize $\mathcal{W}(r,\ t)$ if necessary). Update $\mathcal{Q}$ so as to maintain the $\mathcal{Q}$-invariant over this change to $\mathcal{W}(r,\ t)$.
   (v) If z is $\langle r,\ P(s,t),\ Q(t,\ u)\rangle$, assert the corresponding conclusion of r and move $P(s,t)$ from the P-block preceding $Q(t,u)$ to the one following and update $\mathcal{Q}$ so as to maintain the $\mathcal{Q}$-invariant.

It is easy to check that the saturation procedure is sound — each state transition corresponds to the invocation of a highest priority rule instance. It is also easy to check that the procedure does not terminate until a saturated state has been reached. We now consider the running time of the saturation procedure. We assume that all expressions are "interned", also known as "hash consed", so that the same expression is always represented by the same pointer to memory. This allows equality testing to be done in unit time. Assuming that hash table operations take unit time, matching and instantiating a given pattern can also be done in unit time. Computing the priority of a pair $\langle r,\ \sigma\rangle$ or $\langle r,\ A\rangle$ can also be done in unit time. Let k be the lowest priority (largest number) of any fixed priority rule in the given program. We implement $\mathcal{Q}$ as a pair of a priority queue for items of priority k or less and a Fibonacci heap for higher priorities. For priorities of k or less the operations of insertion, deletion, and merging of queues all take unit time. For larger priorities the Fibonacci heaps support $O(1)$ amortized insertion and queue merger and $O(\log N)$ amortized removal and finding the highest priority element.[2] The P-blocks are implemented as doubly linked lists for fixed priority rules and as Fibonacci heaps for variable priority rules.

First we show that the total amount of time spent inside the assert procedure is $O(|D| + P_f + (P_v + A_v)\log N)$. Each call to the assert procedure either

[2] The amortized time for removal and find-min operations in fibonacci heaps is usually given as $O(\log n)$ where n is the number of elements on the queue. By using at most one element for each priority, where that element is a doubly linked list of "subelements", we can implment removal and find-min in $O(\log N)$ amortized time where N is the number of distinct priorities.

corresponds to an element of D or to a unique strong prefix firing corresponding to the value of z in step (i) of the saturation procedure. Since we are assuming that expressions are interned we can determine whether an expression is already a member of S by checking for an appropriate field in the data structure for that assertion. So asserting an expression that is already visible takes $O(1)$ time. For a new assertion we consider the cost of each step in the assert procedure. Step 1 involves simply setting the value of an appropriate field in the data structure for the assertion and takes $O(1)$ time. To analyze the cost of removing B in step 2 we note that for any given assertion B and rule r there is at most one triple of the form $\langle r,\ A,\ B\rangle$ and at most one triple of the form $\langle r,\ B,\ A\rangle$ in $\mathcal{Q}$. This implies that the total number of removals in step 2 is $O(1)$. For variable priority rules, a single removal costs $O(\log N)$ (amortized) including the time spent merging P-blocks and updating $\mathcal{Q}$ to maintain the $\mathcal{Q}$-invariant. However, a single assertion can only be removed once and hence the total time spent removing instances of antecedents of variable priority rules is $O(A_v \log N)$. So the total time in step 2 is $O(|D| + P_f + P_v + A_v \log N)$. We now consider step 3 of the assert procedure. Since Fibonacci heaps support $O(1)$ time (amortized) insertions, the total time in step 3 is $O(D + P_f + P_v)$. A similar observation applies to step 4.

Finally we consider the time spent executing the saturation procedure not counting time spent inside calls to assert. The time for steps 1 and 2 is $O(D)$. Each iteration of step 3 corresponds to a strong prefix firing of some rule. Step (i) of 3 takes $O(1)$ time if the rule involved is constant priority and $O(\log N)$ time if the rule involved is variable priority. So the total time taken by the removals in substep (i) is $O(P_f + P_v \log N)$. Substeps (ii) and (iii) are $O(1)$ time per iteration. We now consider substep (iv). Insertion into a P-block takes $O(1)$ time (amortized). This insertion may involve a removal and an insertion into $\mathcal{Q}$ in order to maintain the $\mathcal{Q}$-invariant. The insertion takes $O(1)$ but in the case where the rule is variable-priority the removal can take $O(\log N)$ time. Fortunately, in this case the prefix firing corresponding to the value of z is a variable-priority firing. So the time taken in step (iv) is $O(P_f + P_v \log N)$. A similar analysis holds for substep (v). We now have the total execution time is $O(|D| + P_f + (P_v + A_v) \log N)$.

5 Conclusion

We have given a broad scope algorithmic logic programming model of computation. The increased scope of the model over those given in [5] and [2] has come at the cost of some increased complexity in both the semantics of the language and in the associated notion of abstract running time for logic programs. We are hopeful, however, that this language has sufficient scope to cover the majority of algorithms found in standard texts. It would indeed be interesting to write an algorithms text based on an algorithmic logic programming model of computation.

References

1. Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996. 210
2. H. Ganzinger and D. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proc. International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer-Verlag, 2001. 210, 214, 215, 222
3. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press. 210
4. Kolaitis. The expressive power of stratified logic programs. *INFCTRL: Information and Computation (formerly Information and Control)*, 90, 1991. 210
5. David McAllester. The complexity analysis of static analyses. In *Symposium on Static Analysis*. Springer Verlag, 1999. 209, 222
6. Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*. MIT Press, 1991. 210
7. Jeff Polakow. Ordered linear logic and applications, 2001. Ph.D. Dissertation, Carnegie Mellon University. 210
8. T. Przymusinski. the declarative semantics of stratified deductive databases and logic programs, 1988. 210
9. K. Sagonas, T. Swift, and D. S. Warren. Xsb as an efficient deductive database engine. In *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, 1994. 210
10. H. Tamaki and T. Sato. Old resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, 1986. 210
11. J. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eigth ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 140–149, March 1989. 210
12. Jeffrey Ullman and Raghu Ramakrishnan. A survey of research in deductive database systems. *J. Logic Programming*, pages 125–149, May 1995. 210
13. M. Vardi. Complexity of relational query languages. In *14th Symposium on Theory of Computation*, pages 137–146, 1982. 210
14. Yan Zhang and Norman Y. Foo. Towards generalized rule-based updates. In *IJCAI-97*, 1997. 210

Logical Loops

Joachim Schimpf

IC-Parc, Imperial College
London SW7 2AZ, United Kingdom
J.Schimpf@ic.ac.uk

Abstract. We present a concrete proposal for enhancing Prolog and Prolog based Constraint Logic Programming languages with a new language construct, the logical loop. This is a shorthand notation for the most commonly used recursive control structure: the iteration or tail recursion. We argue that this enhancement fits well with the existing language concepts, enhances productivity and maintainability, and helps newcomers to the language by providing concepts that are familiar from many other programming languages. The language extension is implemented and has been in everyday use over several years within the ECL^{i}PS^{e}system.

1 Introduction

Almost 30 years after its inception, Prolog and Prolog-based systems are still the most widespread implementations of logic programming languages. Prolog exhibits a characteristic simplicity, and economy of concepts, which makes some fall in love with the language, but confuses many newcomers, and prompts others not to take it seriously.

More recent attempts at better logic programming languages have taken quite radical measures, like adding strict typing and moding (Gödel [6], Mercury [9]), integrating logic and functional styles (Curry [5], Mercury) or dismissing most of the Prolog heritage, keeping essentially logical variables in a host language that concentrates on other main aspects (Oz [10], Ilog solver [8]).

The enhancements presented in this paper are not about such fundamental modifications. They keep the underlying concepts unchanged, while at the same time making Prolog programming

- more effective and maintainable, by allowing shorter programs and reducing the likelihood of errors
- more elegant and readable, by making the programmer's intention more explicit
- more accessible to newcomers, by providing familiar concepts

The features we introduce can be explained, understood, and implemented by program transformation or preprocessing. They could, therefore, be dismissed as mere 'syntactic sugar'. However, we believe that the substantial benefit gained from such comparatively simple measures warrants this presentation.

For a new language feature to make a difference and become fully accepted by programmers, there are three prerequisites:

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 224–238, 2002.
© Springer-Verlag Berlin Heidelberg 2002

1. it has to fit naturally with the existing language concepts as well as the programmer's preconceptions
2. it has to provide a clear advantage when used (be it in code size, elegance, maintainability, robustness or otherwise)
3. it must not have an overhead cost when used (otherwise programmers will be tempted to use more efficient, lower level methods)

The enhancements described here are all implemented and have been part of the ECL^{i}PS^e constraint logic programming system [4] since 1998. They are and have been in everyday use by a substantial number of programmers and are, to the author's best knowledge, well accepted and appreciated by the users of the system. Although we report experiences with a constraint logic programming system, rather than plain Prolog, this distinction is not essential for our presentation. The stronger declarative character of constraint logic programming does however make the proposed features even more attractive in that setting.

The rest of the paper is divided into motivation, introduction of the loop construct, comparison with higher-order approaches, comparison with bounded quantification, and a discussion of some remaining semantic issues.

2 Motivation

Our work was motivated by our observation of different classes of Prolog users. First, newcomers to Prolog, who already struggle with the unfamiliar concepts of logical variables and backtracking, are uncomfortable with the requirement to do everything by recursion - the hope was that an iteration construct would lower the threshold for them. Second, in our particular area of research, we had a need to convince mathematical programmers that one could model mathematical optimization problems in Prolog - the loop construct (together with additional support for arrays) helped us in this respect. Third, there are well-known software engineering issues when Prolog is being used for larger applications. Although the features described here present only a part of our efforts to address those issues, they do make a contribution by helping to make programs more readable, understandable and thus easier to maintain.

3 Loops

A look at the average Prolog program shows that the vast majority of all recursions are in fact iterations. Most of them are iterations over lists, some are iterations over integers (which may or may not represent structure/array indices), and very few iterate over other structures or index sets.

Not only novice programmers, and programmers that have been trained on imperative languages, resent to being forced to express everything through recursion. Seasoned Prolog programmers also find it tedious to have to write an auxiliary predicate for every iteration. We note that these auxiliary predicates often serve no other purpose: they are only invoked once, they are not useful on

their own, they are not abstractions of any useful concept. Incidentally, this often makes it difficult to invent sensible names for these predicates: in practice they often just inherit the parent predicate's name adorned with some suffix.

Our initial idea for improving the situation was to provide a comprehensive library of higher-order primitives (see e.g. [7]). A limited version of such a library had been available before, providing efficient versions of the basic higher-order predicates like map/3, foldl/4, filter/3 etc. However, this library was under-used and never seemed to provide quite the right tool. During the redesign, we soon realised that going beyond these basics would require to consider additional concepts like lambda expressions, composition of higher-order predicates, and program transformation for efficiency. This however, would have conflicted with our initial objective: it was unlikely that anything based on such complex concepts would be readily accepted by novices, and it was doubtful whether as such it would have constituted a simplification compared to the recursions we wanted to replace.

3.1 Iteration

Our eventual solution is more easily explained as a shorthand for common programming patterns (we will discuss its relationship with the higher order approach later in section 6). Given the importance of iteration, and the ubiquity of loop constructs in imperative languages, why should it not be possible to come up with a loop construct that would take into account the particularities of logic programming?

Some requirements were clear. We definitely wanted to be able to replace the verbose and tedious

```
write_list(List) :-
    write("List: "),
    write_list1(List).

write_list1([]).
write_list1([X|T]) :-
    write(X),
    write_list1(T).
```

with something straightforward like

```
write_list(List) :-
    write("List: "),
    ( foreach(X,List) do write(X) ).
```

This can obviously be implemented quite easily by automatically generating a recursive auxiliary predicate from the do-loop construct, and replacing the do-construct with a call to this auxiliary.

Similarly, we would like to iterate over the arguments of a structure by writing

```
?- ..., ( foreacharg(X,Structure) do write(X) ).
```

or over consecutive integers by writing

```
?- ..., ( for(I,1,100) do write(I) ).
```

3.2 Aggregation

Iteration is usually not done to perform side effects like in the above example, but to accumulate information, for instance compute the sum of list elements. In Prolog, one would use an accumulator pair of arguments to a recursive predicate as in

```
?- ..., sumlist(Xs, 0, Sum).

sumlist([], S, S).
sumlist([X|Xs], S0, S) :- S1 is S0+X, sumlist(Xs, S1, S).
```

In our shorthand notation, we introduce the fromto-specifier which is used in the following way:

```
?- ..., ( foreach(X,Xs),fromto(0,S0,S1,Sum) do S1 is S0+X ).
```

The intuition is that the aggregation process starts *from* the constant `0` and eventually gets *to* the resulting `Sum`. In between, each individual iteration step starts from `S0` (the result that has been accumulated so far) and computes the next intermediate result `S1`.

3.3 Mapping

Mapping (in the sense of the higher-order predicate map/3) means to establish a relationship between two lists (or arrays) by stating that a particular relationship holds between all pairs of corresponding list elements. Corresponding elements are those that occupy the same position in their respective list. Mapping is an extremely common concept in logic programming.

In Prolog, mapping is achieved in a straightforward way by recursing over two lists in parallel:

```
?- ..., one_up(Xs, Ys).

one_up([], []).
one_up([X|Xs], [Y|Ys]) :- Y is X+1, one_up(Xs, Ys).
```

Unlike in functional programming, mappings in Prolog have the nice property that they work in multiple modes (as long as the predicate establishing the relationship between the elements works in multiple modes). This means they can be used to test a relationship between two lists, to construct the second list given the first, construct the first list given the second, or even generate all valid pairs of lists.

Mapping can be covered simply by extending our loop syntax to allow iteration over two lists at the same time, allowing us to write the above example as

```
?- ..., ( foreach(X,Xs),foreach(Y,Ys) do Y is X+1 ).
```

Obviously, every *foreach* specifier corresponds to one argument in the recursive formulation. It is therefore a simple generalisation to allow an arbitrary number of *foreach* specifiers in one loop, this allowing iteration over many lists in parallel. In terms of higher-order predicates, this corresponds to map/4, map/5 etc. For example the above example can be generalised to

```
?- ..., ( foreach(X,Xs),foreach(Y,Ys),foreach(Z,Zs) do Z is X+Y ).
```

3.4 Full Functionality

The ideas introduced above are the basic ingredients for our general loop construct: It has a close correspondence to a simple tail-recursive predicate, and it can have one or more *iteration specifiers*, each of which corresponds to one or two arguments in a recursive predicate. The general form of a logical loop is

(***IterationSpecs*** `do` ***Body***)

where *IterationSpecs* is a comma-separated sequence of iteration specifiers, and *Body* is a general goal (possibly compound). Valid iteration specifiers (in our actual implementation) and their informal meanings are

fromto(First,In,Out,Last) Iterate *Body* starting with In=First and stopping with Out=Last. In and Out are local variables in *Body*.

foreach(X,List) Iterate *Body* with X ranging over all elements of List from first to last. X is a local variable in *Body*. This can be used both for iterating over an existing list or for constructing a new list.

foreacharg(X,StructOrArray) Iterate *Body* with X ranging over all arguments of StructOrArray from left to right. X is a local variable in *Body*. Cannot be used for constructing a term.

for(I,MinExpr,MaxExpr) Iterate *Body* with I ranging over numbers from MinExpr to MaxExpr. I is a local variable in *Body*. MinExpr and MaxExpr can be arithmetic expressions. Can be used only for controlling iteration, i.e. MaxExpr cannot be uninstantiated.

for(I,MinExpr,MaxExpr,Step) The same as above, but a step width different from 1 can be specified.

count(I,Min,Max) Iterate *Body* with I ranging over ascending integers from Min up to Max. I is a local variable in *Body*. This is similar to the for-specifier, but its main use is for counting iterations rather than controlling them, i.e. Max can be uninstantiated.

param(Var1,Var2,...) For declaring variables in *Body* as shared with the context. By default, variables in *Body* are local. For a more detailed discussion see section 8.

In principle, the *fromto* specifier alone would be sufficient: it is the most general one and can be used to express all the others (which we leave as an exercise for the interested reader). On the other hand, one could have introduced even more specifier shorthands, for instance a reverse list iterator, or a list suffix iterator. We have settled with the above set because it provides reasonably intuitive shorthands for the most common cases, in addition to the general *fromto*.

4 Transformation Scheme

We now give the precise semantics of our loop construct by transformation to plain Prolog. Every goal of the form

(*IterationSpecifiers* do *Body*)

is substituted by a goal

PreCallGoals, ℓ(*CallArgs*)

where ℓ is a new, unique predicate symbol, ***CallArgs*** is a sequence of arguments to ℓ, and ***PreCallGoals*** is a possibly empty conjunction of goals to be executed before the call to ℓ. In addition, the transformation creates a definition for ℓ which is always of the following form[1]:

ℓ(***BaseArgs***) :- !.
ℓ(***HeadArgs***) :- ***PreBodyGoals***, ***Body***, ℓ(***RecArgs***).

Here, ***BaseArgs***, ***HeadArgs*** and ***RecArgs*** are sequences of arguments, ***PreBodyGoals*** is a possibly empty conjunction of goals, and ***Body*** is a literal copy of the original loop body.

Figure 1 shows a detailed tabular rendering of the translation rules. The translation algorithm looks up the matching rule for every specifier and collects each specifier's contribution to the six argument/goal sequences. Finally, a loop replacement goal and an auxiliary predicate definition are assembled from these collected sequences. The order of the specifiers in the do-construct is not important[2]. It is merely a specification of what the loop iterates over.

Example Consider the following loop with three iteration specifiers:

```
?- ( foreach(X,List), count(_,1,N), fromto(0,S0,S1,Sum) do
       S1 is S0+1
   ).
```

According to the above specification, the transformation results in the following goal and predicate definition:

```
?-  From is 1-1,                    % PreCallGoals
    do_1(List, From, N, 0, Sum).    % Initial call

do_1([], _1, _1, _2, _2) :- !.      % Base clause
do_1([X|_1], _2, _3, S0, _4) :-     % Recursive clause head
    I is _2 + 1,                    % PreBodyGoals
    S1 is S0+1,                     % Body
    do_1(_1, I, _3, S1, _4).        % Recursive call
```

[1] The cut (!) in the definition should be ignored for now, its role is discussed in detail in section 5

[2] Although with some compilers the indexing in the auxiliary may be affected

Translation Scheme for Iteration Specifiers

Iteration Specifier ***IterationSpecifier***	Transformation-time condition	Pre-Call goals ***PreCallGoals***	Initial call arguments ***CallArgs***	Base clause head arguments ***BaseArgs***	Recursive clause head arguments ***HeadArgs***	Pre-Body goals ***PreBodyGoals***	Recursive call arguments ***RecArgs***
fromto(***From***,***I0***,***I1***,***To***)	nonground(***To***)		***From***, ***To***	L0, L0	***I0***, L1		***I1***, L1
fromto(***From***,***I0***,***I1***,***To***)	ground(***To***)		***From***	***To***	***I0***		***I1***
foreach(***X***,***L***)			***L***	[]	[***X***|T]		T
foreacharg(***A***,***S***)		functor(***S***,_,N), N1 is N+1	***S***, 1, N1	_, I0, I0	***S***, I0, I2	I1 is I0+1, arg(I0,***S***,***A***)	***S***, I1, I2
count(***I***,***FromExpr***,***To***)	var(***I***), integer(***To***)	From is ***From-Expr***−1	From	***To***	I0	***I*** is I0+1	***I***
count(***I***,***FromExpr***,***To***)	var(***I***)	From is ***From-Expr***−1	From, ***To***	L0, L0	I0, L1	***I*** is I0+1	***I***, L1
for(***I***,***FromExpr***,***To***)	var(***I***), number(***To***), Stop is ***To***+1	From is min(***FromExpr***, Stop)	***From***	Stop	***I***	I1 is ***I***+1	I1
for(***I***,***FromExpr***,***ToExpr***)	var(***I***)	From is ***From-Expr***, Stop is max(From, ToExpr+1)	From, Stop	L0, L0	***I***, L1	I1 is ***I***+1	I1, L1
param(***P***)			***P***	***P***	***P***		***P***

Fig. 1. Translation scheme for Iteration Specifiers

The meta-level variables in bold italic style (e.g.***From***) stand for arbitrary terms occurring in the source program. The symbols T, N1, I0, I1, I2, L0, L1, Stop represent auxiliary variables which get introduced by the transformation (new instances for every iteration specifier). Specifiers which do not match any of these rules are treated as compile-time errors. The translation of the for/4 specifier has been omitted for space reasons.

Implementation In our system, this transformation is normally performed by the inlining facility of the compiler. In most Prolog systems, a similar effect can be achieved by means of the *term_expansion* mechanism.

In case a do-loop is constructed at runtime and meta-called, the system performs the same transformation, but meta-calls (i.e. interprets) the resulting code rather than actually generating a recursive predicate. One one hand, this is based on the guess that the compilation overhead might outweigh the gains when the loop is only run once, on the other hand this avoids the generation of an unbounded number of auxiliary predicates, and related garbage collection issues.

5 Loops vs. Recursion

As opposed to the equivalent recursive formulation, the loop construct has a number of advantages which as (not necessarily in order of importance):

Conciseness No need to write an auxiliary predicate, in particular no need to invent a name for the recursive predicate, and no need to worry about the arity of the recursive predicate. This leads to 2-3 times shorter code (in terms of token count) in the above examples[3].

Modifiability If an additional value needs to be computed by the iteration, rather than having to add an argument or an accumulator pair in 4 places in the code (call, base clause, recursive clause head, recursive call), a single iteration specifier is added to the loop.

Structure Loops can be freely nested. This will usually show the code structure more clearly than a flat collection of predicates. Also, the iteration specifiers group conceptually related information better than scattered predicate arguments.

Abstraction An iteration specifier is an abstraction for a single induction argument or an accumulator pair. For example in this efficient list reversal predicate

```
reverse(L, R) :-
   ( fromto(L,[X|Ls],Ls,[]),fromto([],Rs,[X|Rs],R) do true ).
```

the first fromto translates into a single argument of do_2/3, while the second translates into an argument pair:

```
reverse(L, R) :- do_2(L,[],R).
do_2([],R,R) :- !.
do_2([X|Ls],Rs,R) :- do_2(Ls,[X|Rs],R).
```

The programmer does not need to be concerned about this detail. Both fromto-specifiers look completely symmetric, they graphically specify the order (from L to [], and from [] to R) in which the two lists are being traversed. In fact the predicate works both ways.

[3] assuming the common case that no recursive predicate was called more than once

Usability Although we have only anecdotal evidence, loops have clearly become very popular among the users of our implementation. Fears, that the additional feature would confuse new users more than it helped, seem to have been unjustified. On the contrary, it seems that loops with fromto-specifiers help with understanding the equivalent concept of accumulator pairs in recursive code.

6 Loops vs. Higher-Order Constructs

Our loop construct offers an alternative to three of the most commonly used higher-order programming constructs which have found their way from functional programming into logic programming: map/3, foldl/4 and filter/3 (cf. [7]). For instance:

```
map(plus(1),Xs,Ys)      ( foreach(X,Xs), foreach(Y,Ys) do
                              plus(1, X, Y)
                        )

foldl(plus,Xs,0,Sum)    ( foreach(X,Xs), fromto(0,S0,S1,Sum) do
                              plus(X,S0,S1)
                        )

filter(<(5),Xs,Ys)      ( foreach(X,Xs), fromto(Ys,Ys1,Ys0,[]) do
                              ( X > 5 -> Ys1=[X|Ys0] ; Ys1=Ys0 )
                        )
```

In those simple instances, the loop formulation is somewhat more verbose than the higher order one. The reason for this is that the higher order formulation relies on an auxiliary predicate with a fixed argument convention (e.g. plus/3 with the last two arguments being the input and output of the mapping). Except in lucky circumstances, this auxiliary predicate will have to be purpose-written for each use, a development overhead that we would rather avoid.

Let us therefore consider a more flexible higher-order formulation with lambda-terms (a syntax for anonymous predicates similar to the one introduced in [12]), which would avoid the need for the auxiliary, e.g.

```
foldl(lambda([X,S0,S1], S1 is S0+X), Xs, 0, Sum)
```

This formulation is now not only of the same length, but also structurally very similar to our loop formulation. In fact there is a one-to-one correspondence of constants and variables:

```
foreach(X,Xs), fromto(0,S0,S1,Sum) do S1 is S0+X
```

Which version is preferable to a programmer is partly a matter of taste and will depend on training and on experience with other languages and programming paradigms. It can however be argued that the loop formulation exhibits a clearer grouping of related items: the variables X and Xs which are related to the list

iteration aspect, and the variables S0,S1,Sum together with the constant 0 which are related to the aggregation aspect of the code fragment.

A disadvantage of the higher-order constructs is the need for more and more: map/4, map/5, and combinations like map_foldl/5, etc. are frequently needed, but the provision of all these special cases can only be avoided by sophisticated program transformation, or the use of auxiliary data tuples [7]. In contrast, the same loop construct can be used for all these generalisations and combinations.

While it is clear that the higher-order approach has other uses, we argue that, for expressing iterations, our loop construct is preferable because

1. the loop construct can replace the vast majority of recursions in a form that is similarly compact and at the same time more explicit in stating the programmer's intent. Iteration seems an important enough concept to warrant a special language construct.
2. higher-order constructs implement arbitrary (not necessarily iterative) traversals of a particular data structure, while loops implement only iterative traversals but over arbitrary data structures.
3. the loop construct encourages the use of an efficient form of recursion, viz. tail recursion. This contrasts with the higher order approach which makes the inefficient foldr and the efficient foldl look interchangeable.
4. the single loop language construct covers the ground of several higher-order predicates families: map/(2+N), foldl/(2+2N), filter/(2+N).
5. the loop construct provides the building blocks to formulate arbitrary combinations of map / foldl / filter.
6. arguably, both novices and experienced programmers have fewer problems reading and understanding the meaning of a loop than they have understanding the meaning of a higher-order formulation.
7. unlike the higher-order solution, no complex higher-order typing is involved.

7 Loops vs. Bounded Quantification

Voronkov [11], Barklund and Bevemyr [2], Barklund and Hill [3], as well as Apt [1] have advocated the introduction of bounded quantifiers. Their motivation is similar to ours: to express iteration more concisely, and in a way that is often closer to the original specification.

A bounded quantification requires a single finite set (e.g. an integer range, the elements of a list, etc.) over which the quantification ranges. But while [2] and [1] consider only the case where the quantification is bounded *a priori*, [11] is more general in allowing the termination condition to be depend on the quantified formula itself. The latter is what makes Voronkov's language Turing-complete, even without recursion.

Although bounded quantifiers do provide a significant gain in expressive elegance, many simple tasks cannot be expressed at all using *a priori* bounded quantification, and are still difficult to express in Voronkov's more powerful language. Consider the simple problem of determining that two lists are identical (or satisfy any other mapping property in the sense of map/3). This cannot be

expressed with *a priori* bounded quantification because there is no way to express which list elements correspond to each other. The obvious workarounds that come to mind are either to quantify over a list of pairs, or to convert the lists to arrays and then to quantify over the array index. However, this just raises the equally unsolvable problems of how to construct a list of pairs from two simple lists, or of how to construct an isomorphic array from a list.

With Voronkov's quantifiers the formulation is of course possible, but the best we could come up with is the following rather unnatural solution, employing the list-suffix-quantifier:

```
same_lists(XXs, YYs) :-
   SameTails = [XXs-YYs|_],
   (∀ T ⊑ SameTails) T = [[]-[]] ∨ T = [[X|Xs]-[X|Ys],Xs-Ys|_].
```

The loop construct overcomes this problem simply by having a concept of implicitly ordered iteration steps, and by allowing multiple iteration specifiers to synchronously traverse multiple data structures or index ranges[4]:

```
same_lists(XXs, YYs) :-
    ( foreach(X,XXs), foreach(Y,YYs) do X=Y ).
```

It can be argued that these difficulties with expressing mappings in the quantifier approach are partially overcome (or obscured) by the use of arrays. When arrays are used instead of lists, the above example is easily expressed, by quantification over a common array index. Index positions are a way to establish mappings explicitly, but work only with array-like data structures. It is therefore no coincidence that all the work on *a priori* bounded quantification has found it necessary to introduce arrays as a supporting feature.

Iteration specifiers in loops also play the role of the aggregation operators employed by [2], [3] and [1]. Without aggregation operators, *a priori* bounded quantifiers are very limited in their expressive power. Note again that Voronkov [11] does not need aggregation operators, because his quantifiers (like our iteration specifiers) can play this role as well.

In our loops, the general *fromto*-specifier, the list iterator *foreach* and the integer iterator *count* can all serve either as quantifiers (controlling the iteration) or as aggregators. We have deliberately avoided to define additional specifiers that would serve only as aggregators, because arbitrary aggregators can be so easily expressed using the general *fromto*. For example, the arithmetic maximum-aggregator is

```
maxlist([X0|Xs], Max) :-
    (   foreach(X,Xs), fromto(X0,M0,M1,Max) do
        ( X > M0 -> M1 = X ; M1 = M0 )
    ).
```

[4] We suggest that the more mathematically inclined reader pronounces all occurrences of the word 'do' in our code as 'holds' or ':' in order to eliminate the procedural taste

The cited works on bounded quantification investigate not only universal quantifiers but also existential quantifiers. We have not addressed this issue here at all, but it would seem that, unless one provides means to choose different control strategies for every existential quantifier, these do not give any significant advantage over the use of member/2 for existential quantification over list elements, or between/3 for existential quantification over integers.

8 Remaining Issues

Variable Scope in Loop Bodies One aspect that can potentially cause confusion is that the loop body is really an embedded predicate body with its own local variable scope. The following incorrect code illustrates this. The two occurrences of 'Array' are different variables:

```
sum_array(Array, N, Sum) :-
    ( for(I,1,N), fromto(0,S0,S1,Sum) do
        arg(I, Array, Elem), S1 is S0 + Elem
    ).
```

The programmer instead has to write

```
sum_array(Array, N, Sum) :-
    ( for(I,1,N), fromto(0,S0,S1,Sum), param(Array) do
        arg(I, Array, Elem), S1 is S0 + Elem
    ).
```

This situation has no direct counterpart in normal Prolog. The closest analogy are the bagof/setof predicates which allow locally quantified variables[5]. Our reason to opt for the opposite default (variables are quantified locally inside the loop, unless passed as *param*) was that this makes the loop semantics independent of the context (the presence or absence of a variable in the loop context will not affect the loop semantics). This not only relieves the compiler from the need to analyze the loop context, it also makes meta-calling of loops feasible.

The programmer needs to be aware that every iteration of the loop corresponds to a new instance of the loop body. Given the single assignment property of Prolog variables, this seems to be sufficiently intuitive, and our experience suggests that programmers do not have a problem with this. The only problem that does arise in practice is that the programmer forgets to specify the global variables. Fortunately, in many cases this situation leads to singleton variables in the loop body and our compiler gives a warning, suggesting that a *param* might be missing.

Nondeterminism Another point we have glossed over in the above is that our loop transformation, as implemented, always puts a cut into the base clause, i.e. the transformation template for the recursive predicate is in fact

[5] At the expense of a considerable implementation overhead

```
aux(...) :- !.
aux(...) :- ..., aux(...).
```

Most of the time, this makes no difference to the semantics (i.e. it is a so-called *green cut*). It does of course prevent applications where the number of iterations is nondeterministic and increases on backtracking.

Nevertheless we consciously made this restriction. The reason was that we would have been carried too far away from the intuitive idea of a loop. A choicepoint left by the loop construct itself would normally be unexpected and most likely constitute a (hard to find) bug. This is in fact analogous to Prolog's if-then-else construct `(... -> ... ; ...)`, where choicepoints within the condition are also cut, for very similar reasons.

Note that, of course, the loop body can be nondeterministic and generate multiple solutions. All we prevent is the number of iterations being nondeterministic.

Termination The termination condition of our loops is restricted to unification, or a conjunction of unifications. Loops that are terminated by a more complex condition can be expressed, but only indirectly.
Consider the iteration pattern:

```
p(... X0 ...) :-
    ( termination_condition(X0) ->
        true
    ;    ...,
        p(... X1 ...)
    ).
```

This can be expressed through a loop by introducing an explicit control variable `Continue`:

```
(
    fromto(continue, _, Continue, stop),
    fromto(..., X0, X1, ...), ...
do
    ( termination_condition(X0) ->
        Continue = stop
    ;   ...,
        Continue = continue
    )
)
```

This is rather unnatural, and in such cases the use of the loop construct will often not be appropriate. The example also shows that termination of loops with fromto-specifiers is not decidable in the general case. But in many special cases (where iteration specifiers correspond to *a priory* bounded quantifiers) termination is trivially guaranteed. This is true for the *foreacharg* and the *for* specifier, and for the *foreach* specifier when the list position is instantiated to a proper list.

Computational Power Prolog with logical loops, but without recursion, is still Turing-complete. The following is a recursion-free meta-interpreter for pure Prolog:

```
solve(Q) :-
    ( fromto([Q], [G|C0], C1, []) do solve_step(G, C0, C1) ).

solve_step(true, C, C).
solve_step((A,B), C, [A,B|C]).
solve_step(A, C, [B|C]) :- clause(A, B).
```

Typing Unlike most of the related work, we have not found it inevitable to introduce typing into our language. In the bounded quantifier framework, typing is used for specifying the semantics, in particular the domains over which the quantifiers range. In our loop framework, the semantics is formally defined by way of program transformation, which does in itself not provide a motivation for typing.

9 Conclusion

We have presented an addition to the Prolog programming language that makes programs more concise, more readable, easier to modify, less error-prone and more accessible to newcomers. Iteration often makes it possible to express a problem in a way that is closer to the original problem specification and also closer to the programmer's intuition.

We have not introduced any fundamental change to the language. In particular, we have neither introduced typing nor a concept of function evaluation that goes beyond what is already present in the basic language. Our loop construct can be entirely specified in terms of preprocessing, but is also easy to understand directly.

We have argued that our proposal is closely related to certain well-known higher-order constructs, but can have advantages over a corresponding higher-order formulation. Similarly, we have looked at the relationship with bounded universal quantifiers and shown that our approach in many cases allows a more natural formulation.

One direction of future work could be to look at ways to compile iterations more efficiently than the equivalent recursion. We would expect the techniques investigated in [2] to be applicable to our language. We intend to make the full loop transformation code available under `http://www.icparc.ic.ac.uk/eclipse/software/loops/`.

Acknowledgements

I would like to thank Mark Wallace and Stefano Novello for many discussions on the subject. Part of the work presented here was done in the context of the CHIC-2 project and I would like to thank our partners, in particular at EuroDecision, for

motivating me to make Prolog more suitable for mathematical modelling. Further thanks to Carmen Gervet, Kish Shen, Josh Singer and Warwick Harvey for their comments on earlier drafts of this paper.

References

1. K. R. Apt. Arrays, bounded quantification and iteration in logic and constraint logic programming. *Science of Computer Programming*, 26(1-3):133–148, 1996. 233, 234
2. J. Barklund and J. Bevemyr. Prolog with arrays and bounded quantifications. In A. Voronkov, editor, *Proceedings of LPAR'93*, pages 28–39. Springer, 1993. 233, 234, 237
3. J. Barklund and P. Hill. Extending Gödel for expressing restricted quantifications and arrays. Technical Report No. 102, Uppsala University, March 1995. 233, 234
4. ECLiPSe Team. ECLiPSe User Manual Version 4.0. Technical report, IC-Parc, Imperial College, London, July 1998. 225
5. M. Hanus. Curry: An integrated functional logic language. Technical report, Univerity of Kiel, Kiel, Germany, June 2000. 224
6. P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, 1994. 224
7. L. Naish. Higher-order logic programming in Prolog. Technical Report 96/2, University of Melbourne, Feb. 1996. 226, 232, 233
8. J.-F. Puget. A C++ implementation of CLP. In *Proceedings of SPICIS 94*, Singapore, November 1994. 224
9. Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995. 224
10. P. Van Roy. Logic programming in Oz with Mozart. In D. D. Schreye, editor, *International Conference on Logic Programming*, pages 38–51, Las Cruces, NM, USA, Nov. 1999. The MIT Press. 224
11. A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming, First and Second Russian Conference*, pages 486–514. Springer LNAI, 1990/1991. 233, 234
12. D. H. D. Warren. Higher-order extensions to Prolog - are they needed? *Machine Intelligence*, 10:441–454, 1982. 232

Learning in Logic with RichProlog

Eric Martin[1], Phuong Nguyen[1], Arun Sharma[1], and Frank Stephan[2]

[1] School of Computer Science and Engineering, The University of New South Wales
Sydney, NSW 2052, Australia
{emartin,ntp,arun}@cse.unsw.edu.au
[2] Universität Heidelberg, 69121 Heidelberg, Germany
fstephan@math.uni-heidelberg.de

Abstract. Deduction and induction are unified on the basis of a generalized notion of logical consequence, having classical first-order logic as a particular case. RichProlog is a natural extension of Prolog rooted in this generalized logic, in the same way as Prolog is rooted in classical logic. Prolog can answer Σ_1 queries as a side effect of a deductive inference. RichProlog can answer Σ_1 queries, Π_1 queries (as a side effect of an inductive inference), and Σ_2 queries (as a side effect of an inductive inference followed by a deductive inference). RichProlog can be used to learn: a learning problem is expressed as a usual logic program, supplemented with data, and solved by asking a Σ_2 query. The output is correct in the limit, *i.e.*, when sufficient data have been provided.

1 Introduction

Enriching the expressive power of the logical language that subsumes the theory of Logic Programming has long been an active area of research. Examples of such approaches include consideration of negative literals in the body of clauses, interpreted either as negation as failure or as 'true' negation (for surveys see [3,12]), and answering queries more complex than existentially quantified conjunctions of atoms [7]. All these investigations, however, have been in the realm of classical deductive logic. Since the class of Prolog programs determines an acceptable indexing of the class of all partial recursive functions, it might be argued that the fragment of classical logic that deals with rules (definite Horn clauses) and queries is expressive enough for all purposes. This is certainly true for the task of computing; but not for the task of computing by writing programs in a declarative manner — the *raison d'être* of Prolog.

To illustrate the above point, let us consider the task of implementing a learning strategy. Learning is not deducing. Indeed, deductive inferences are compact *i.e.*, can always be made on the basis of a finite set of formulas, whereas learning would be too restrictive if it always meant arriving at a definite conclusion on the basis of a finite set of data and background knowledge. So implementing a learning strategy in Prolog requires the use of heuristics that cannot be viewed as a description of the learning problem, since the latter is in essence nondeductive. Unless, if learning can legitimately be viewed as a particular kind of logical, nondeductive inference.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 239–254, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Indeed, a generalized notion of logical consequence can be defined that, given a theory T, yields a hierarchy of generalized logical consequences of T by alternating deductive (or compact) inferences and inductive (or 'weakly compact') inferences. This notion of generalized logical consequence is actually a function of a number of parameters, one of them being a class $\mathcal{W}$ of possible worlds, or intended interpretations. In most applications, a natural choice for $\mathcal{W}$ is the class of Henkin (if equality is allowed) or Herbrand (if equality is not allowed) structures.[1] In classical logic, where $\mathcal{W}$ is the class of all structures, each level of the hierarchy of (generalized) logical consequences collapses to the first level *i.e.*, the level of deductive inferences. But when $\mathcal{W}$ is equal to the class of Henkin or Herbrand structures, a true hierarchy of generalized logical consequences results. It turns out that under natural assumptions, identification in the limit can be characterized as the inference (in the limit) of a generalized logical consequence that belongs to the third level of the hierarchy, *i.e.*, the level of inductive inferences followed by deductive inferences. So a learning problem can be expressed in purely logical terms, and the ideal of programming in logic can be extended to applications such as learning that are beyond the scope of classical logic.

We proceed as follows. We briefly sketch some fundamental concepts of the generalized logic. We describe RichProlog, which is a natural enrichment of Prolog based on a very particular instance of the generalized logic that is capable of answering Σ_2 and Π_1 queries in addition to the usual Σ_1 queries that can answered by Prolog. Finally, we show how RichProlog can be used to learn non-erasing pattern languages from positive data.

The motivation for development of systems capable of answering Σ_2 queries is obvious from the perspective of discovering more sophisticated knowledge. Consider the hypothetical scenario of finding a vaccine for a virus. Let variable y range over different instantiations of a virus. Let $P(x, y)$ describe the property that x "disables" y. P may be a complex relation based on certain geometric and chemical properties between x and y. Then, the search for a vaccine is an answer to the Σ_2 query: $\exists x \forall y P(x, y)$. While this is a hypothetical scenario, a number of problems in drug design, e.g., *pharmacophore identification*, can be expressed as Σ_2 queries.

2 Logical Foundation

2.1 Generalized Logical Consequence

Denote by S a vocabulary without equality and by $\mathfrak{L}$ the set of first-order S-formulas, referred to more simply as *formulas*. Closed formulas will be called *sentences*; atomic formulas or their negations will be called *basic* formulas. We refer to sets of formulas as *theories*. Denote by E a set of formulas, called set of *possible evidence*. Denote by $\mathcal{W}$ a class of S-structures, called class of *possible*

[1] A Henkin structure consists of individuals each of which interprets a closed term. A Herbrand structure is a Henkin structure such that distinct closed terms are interpreted by distinct individuals.

worlds. For all $\mathfrak{M} \in \mathcal{W}$, the *$\mathcal{E}$-diagram of* $\mathfrak{M}$, denoted $D_{\mathcal{E}}(\mathfrak{M})$, is the set of all members of E true in $\mathfrak{M}$. Here is a typical scenario that could be described as a *paradigm* of Formal Learning Theory. (See [5,9,6] for descriptions and investigations of such paradigms.) A member of $\mathcal{W}$, say $\mathfrak{M}$, is chosen. A *learner* f is presented with every initial segment of an infinite enumeration of the diagram of $\mathfrak{M}$ and reacts by outputting members of $\mathfrak{L}$.[2] More formally:

Definition 1. *A* learner *is a mapping*[3] *from* $\mathcal{E}^*$ *into* $2^{\mathfrak{L}}$.

Suppose for instance that S consists of a constant $\overline{0}$, a unary function symbol s, a unary predicate P, and a binary predicate R. Given $n \in \mathbb{N}$, denote by $\overline{n}$ the term obtained from $\overline{0}$ by n applications of s. Set $\mathcal{E} = \{P(\overline{n}) \mid n \in \mathbb{N}\}$. Assume that $\mathcal{W}$ is the class of Herbrand S-structures $\mathfrak{M}$ such that:

- for all $m, n \in \mathbb{N}$, $\mathfrak{M} \models R(\overline{m},\overline{n})$ iff $m \leq n$;
- $\mathfrak{M} \models P(\overline{n})$ for finitely many $n \in \mathbb{N}$.

Choose for $\mathfrak{M}$ the (unique) member of $\mathcal{W}$ such that for all $n \in \mathbb{N}$, $\mathfrak{M} \models P(\overline{n})$ iff $n \leq 10$. The learner f could be presented with every initial segment of the initial sequence $e = (P(\overline{0}), P(\overline{1}), \ldots P(\overline{10}), P(\overline{0}), P(\overline{1}), \ldots P(\overline{10}), \ldots)$. We might expect f to be able to *discover in the limit* that the formula $\psi = P(\overline{10}) \wedge \forall y(R(s(\overline{10}), y) \rightarrow \neg P(y))$ is true in $\mathfrak{M}$. That is, faced with longer and longer initial segments of e, the learner f should be able to stabilize its outputs to ψ. More formally:

Definition 2. *A learner* f identifies a formula φ in the limit in $\mathcal{W}$ *just in case for all* $\mathfrak{M} \in \mathcal{W}$ *and infinite enumerations* $(e_0, e_1, e_2 \ldots)$ *of members of* $D_{\mathcal{E}}(\mathfrak{M})$ *where every member of* $D_{\mathcal{E}}(\mathfrak{M})$ *occurs at least once,* $\mathfrak{M} \models \varphi$ *iff* $\varphi \in f((e_0, \ldots, e_k))$ *for all but finitely many* $k \in \mathbb{N}$.

Let theory $X = \{R(\overline{m},\overline{n}) \mid m \leq n\} \cup \{P(\overline{n}) \mid n \leq 10\}$. Now note that ψ is not a logical consequence of X. Indeed, the requirement—*every* possible evidence true in the underlying world will eventually appear in the enumeration—cannot be expressed in first-order logic. Moreover, ψ is not even a logical consequence of theory $Y = X \cup \{\neg P(\overline{n}) \mid n > 10\}$ because of the *nonstandard* models of Y, models of Y some of whose individuals are not interpreted by a closed term. Still, ψ can be viewed as a 'generalized logical consequence' of X in a 'generalized logic' that is not as rigid as classical first-order logic, because:

- it does not force us to accept structures we do not want to or have good reasons not to consider as possible interpretations;
- it has a minimality principle that captures the requirement that if some possible evidence χ does not belong to a theory T, then $\neg\chi$ should be considered to be true in every intended interpretation of T.

[2] To be extremely precise, an extra symbol # can also appear in such enumerations, meaning 'no datum now.' This becomes necessary if no member of E is true in $\mathfrak{M}$.

[3] For the purpose of introducing the basic concepts of learning theory, whether this mapping should necessarily be computable is inessential. Given a set X, X^* denotes the set of finite sequences of members of X, and 2^X the set of all subsets of X.

The second condition is obviously closely related to circumscription and the closed world assumption [11,8]. The notion of generalized logical consequence that satisfies both conditions above is then formally defined via the following two definitions, where $\subset$ denotes strict inclusion.

Definition 3. *Let $T \subseteq \mathfrak{L}$ and a structure $\mathfrak{M}$ be given. We say that $\mathfrak{M}$ is an* E-minimal model of T in $\mathcal{W}$ *iff $\mathfrak{M}$ is a model of T in $\mathcal{W}$ and for all models $\mathfrak{N}$ of T in $\mathcal{W}$, $D_{\mathcal{E}}(\mathfrak{N}) \not\subset D_{\mathcal{E}}(\mathfrak{M})$.*

Definition 4. *Given $T \subseteq \mathfrak{L}$ and $\varphi \in \mathfrak{L}$, we say that φ is an* E-minimal logical consequence of T in $\mathcal{W}$, *and we write $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$, iff every E-minimal model of T in $\mathcal{W}$ is a model of φ.*

Definition 4 is a particular case of the notion of *preferential satisfaction* introduced in [13]. Getting back to the example above (where the values of $\mathcal{W}$, E, X and ψ have been fixed), it is clear that ψ is an E-minimal logical consequence of X in $\mathcal{W}$. More informally, we say that ψ is a *generalized logical consequence of X*. We refer the reader to [10] for a detailed development of the notion of generalized logical consequence.

2.2 From Generalized Logic to Generalized Logic Programming

Definitions 3 and 4 generalize basic concepts of the theory of Logic Programming. Indeed, assume that $\mathcal{W}$ and E are defined as above. If T is a set of *rules* that contains $\{R(\overline{m},\overline{n}) \mid m \leq n\}$ and if φ is an existential query, then 'φ is a generalized logical consequence of T' can be paraphrased as 'the (unique) minimal Herbrand model of T is a model of φ,' which we all know is equivalent to: φ is a logical consequence of T [4]. When the theory of Logic Programming is concerned with more general kinds of logic programs or more general kinds of queries, the equivalence fails. Since the focus is still on the classical notion of logical consequence, the intended interpretations can no longer be limited to the class of Herbrand structures. Our framework is the exact dual of this approach. We are concerned with more general kinds of logic programs and more general kinds of queries. Since Herbrand models are intended interpretations, the classical notion of logical consequence is inadequate. Consider the same example again. Representing the task of the learner f as a generalized logic program and a generalized query cannot be done directly in the realm of classical logic, because the task of f is not to discover that some formula is a logical consequence of some theory, but is to discover that some formula is a generalized logical consequence of some theory. Hence, it should be possible to represent the task of f directly, naturally and declaratively as a generalized logic program and a generalized query in the realm of the generalized logic we have introduced. More precisely, no generalized query would represent the task of f better than:

$$\varphi = \exists x \forall y (P(x) \wedge (R(s(\overline{x}), y) \rightarrow \neg P(y))).$$

Given the right generalized logic program T, we expect an interpreter to be able to prove that $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$, and as a side effect, that the formula obtained from

φ by removing the existential quantifier and instantiating x with $\overline{10}$, is also a generalized logical consequence of T. In other words, we expect that it is possible to *compute* the least natural number m such that for all $n \in \mathbb{N}$, $\neg P(\overline{n})$ is true in the underlying world if and only if n is greater than m.

We will not formalize here how we define *hierarchies* of generalized logical consequences, but we explain their fundamental features. (See [10] for a formal treatment.) Denote by A a set of formulas that contains E, called set of *possible axioms* (for instance, A can be defined as the set of rules). Call *possible theory* any set of the form $D_{\mathcal{E}}(\mathfrak{M}) \cup X$, where $\mathfrak{M}$ is a possible world and $X \subset \mathcal{A}$ is a set of possible axioms that are true in $\mathfrak{M}$. A hierarchy of generalized logical consequences of T can be defined for every possible theory T. Such a hierarchy reflects the complexity of generalized logical consequence. Basically, the higher a formula φ occurs in the hierarchy built over a possible theory T, the more difficult is the task of discovering that $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$. The first level of the hierarchy, called the Σ_1 level, corresponds to *deductive inference*: from a finite subset of T, it is possible to conclude with certainty that φ is a generalized logical consequence of T. The next level, called the Π_1 level, corresponds to *inductive inference*: from a finite subset of T, it is possible to believe that φ is a generalized logical consequence of T, since some finite subset of T can *refute* this belief with certainty in case $T \not\models^{\mathcal{E}}_{\mathcal{W}} \varphi$. The very principles that define the Σ_1 and Π_1 levels of the hierarchies can be iterated to define higher levels, starting with the Σ_2 level, followed by the Π_2 level, then the Σ_3 level, then the Π_3 level, etc. Under some assumptions, it is possible to relate the syntactic complexity of a formula with its location in the hierarchy of generalized logical consequences, as shown by the following proposition.

Proposition 1. *Suppose that $\mathcal{W}$ is a set of Henkin structures, E contains all basic sentences, and $\mathcal{A} = \mathcal{E}$. Let $n > 0$ be given. For every Σ_n (respect. Π_n) formula φ and possible theory T, if $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$ then φ belongs to the Σ_n (respect. Π_n) level of the hierarchy of generalized logical consequences of T.*

Proposition 1 is just one of many propositions that describe the structure of the hierarchies of generalized logical consequences built over possible theories, under various assumptions. It can be shown that there exists a learner that identifies in the limit a formula φ (in the sense of Definition 2) iff φ belongs to the Σ_2 level of the hierarchy of generalized logical consequences of T, for every possible theory T such that $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$—see [10]. So by Proposition 1, under some assumptions on $\mathcal{W}$, E and A, the class of formulas that can be identified in the limit is precisely the class of Σ_2 formulas. But Σ_2 formulas can represent *learning problems*. For instance, the problem of learning a law from observed data can be expressed by the Σ_2 statement: 'there exists a law ℓ such that for all possible data d, ℓ predicts d iff d is among the data that are eventually observed.' Solving the learning problem is then reduced to computing a witness for ℓ that represents the law to be learned. When the set E of possible observations is not closed under negation, just a subclass of the class of all Σ_2 formulas can be identified in the limit. But rather than developing the logical framework and the connections with learning theory, we will devote the remaining part of this paper to a description

of RichProlog. RichProlog is a natural extension of Prolog that can be used to infer that a sentence of the form $\exists\overline{x}\forall\overline{y}\varphi$, where φ is quantifier free, is a generalized logical consequence of a possible theory T encompassing background knowledge and observations. The result of the computation is a sequence of terms $\overline{t}$ such that $T \models^{\mathcal{E}}_{\mathcal{W}} \forall\overline{y}\varphi[\overline{t}/\overline{x}]$.[4] Solving a learning problem amounts to computing $\overline{t}$.

3 RichProlog

3.1 The Basic Strategy

From now on we assume that $\mathcal{W}$ is a set of Herbrand structures, while E is a set of atomic sentences. This is the natural choice when RichProlog is used to learn from positive data only. We denote by $\widehat{\mathcal{E}}$ the set of members of E and their negations. We assume that the set A of possible axioms is built from some set A of atomic formulas, none of which has a member of E as an instance, with the following property:

> $(*)$ every member of $\mathcal{A} \setminus \mathcal{E}$ is of the form $\alpha_1 \wedge \ldots \wedge \alpha_p \rightarrow \alpha_0$ (also represented as $\alpha_0 \leftarrow \alpha_1 \wedge \ldots \wedge \alpha_p$, or as $\alpha_0 \text{ :- } \alpha_1, \ldots, \alpha_p$) where $p \in \mathbb{N}$, α_0 is a member of A, and $\alpha_1, \ldots, \alpha_p$ are members of $A \cup \widehat{\mathcal{E}}$.

We call members of A *generalized clauses*. Remember that a possible theory is a set of the form $D_{\mathcal{E}}(\mathfrak{M}) \cup X$, where X is a subset of the set of possible axioms A all of whose members are true in $\mathfrak{M}$. We call *generalized logic program* a possible theory of the form $D_{\mathcal{E}}(\mathfrak{M}) \cup X$ where X is a finite set of possible axioms (*i.e.*, generalized clauses)—but $D_{\mathcal{E}}(\mathfrak{M})$ can obviously be infinite. The aim is to make RichProlog show that a Σ_2 formula φ is a generalized logical consequence of a generalized logic program $D_{\mathcal{E}}(\mathfrak{M}) \cup X$, on the basis of sets of the form $D \cup X$ where D is a finite subset of $D_{\mathcal{E}}(\mathfrak{M})$. Larger and larger subsets D of $D_{\mathcal{E}}(\mathfrak{M})$ will be provided to the system (which corresponds to making more and more observations). RichProlog has to infer correctly that φ is a generalized logical consequence of $X \cup D_{\mathcal{E}}(\mathfrak{M})$ when D is large enough and produce the right witnesses for the existentially quantified variables in φ. Note that for all possible theories T and for any possible evidence $\chi \in \mathcal{E}$, either $T \models^{\mathcal{E}}_{\mathcal{W}} \chi$ or $T \models^{\mathcal{E}}_{\mathcal{W}} \neg\chi$. In the first case, χ belongs to T and will eventually be provided to the system. In the second case, negation as failure correctly infers $\neg\chi$ from any finite subset of $X \cup D_{\mathcal{E}}(\mathfrak{M})$. Hence negation as failure represents 'true negation' w.r.t. the notion of generalized logical consequence. Weaker assumptions on E and A are possible and currently under investigation but in this paper, we limit the discussion to generalized clauses as defined above. Given a generalized logic program T and an atomic formula ψ all of whose free variables occur in the disjoint sequences of variables $\overline{x}$ and $\overline{y}$, RichProlog will determine that $\exists\overline{x}\forall\overline{y}\psi$ is a generalized logical consequence of T whenever this is indeed the case. Moreover, when $\exists\overline{x}\forall\overline{y}\psi$

[4] Given a formula ψ, $n \in \mathbb{N}$, a sequence $\overline{x} = (x_0, \ldots, x_n)$ of distinct variables, and a sequence $\overline{t} = (t_0, \ldots, t_n)$ of terms, $\psi[\overline{t}/\overline{x}]$ denotes the result of simultaneously substituting in ψ every occurrence of x_i by t_i, for all $i \leq n$.

is a generalized logical consequence of T, RichProlog will output a sequence of terms $\overline{t}$ of the same length as $\overline{x}$, a witness for $\exists\overline{x}\forall\overline{y}\psi$, such that $T \models^{\mathcal{E}}_{\mathcal{W}} \forall\overline{y}\psi[\overline{t}/\overline{x}]$. More complex Σ_2 queries can also be tackled, as will be seen in the Section 3.3. Usually T is infinite (because the E-diagram of a possible world is infinite), and RichProlog's outputs are correct *in the limit* that is, from the time when a large enough finite subset of T is available. For the kind of application that will be discussed in this paper, RichProlog's search strategy proceeds in the following two stages.

Stage 1: Choose $n \in \mathbb{N}$ and sequences of terms $\overline{t_1}, \ldots, \overline{t_n}$ of the same length as $\overline{y}$, and find a witness $\overline{t}$ for the query $\exists\overline{x}(\psi(\overline{x}, \overline{t_1}) \wedge \ldots \wedge \psi(\overline{x}, \overline{t_n}))$?

Stage 2: Try to *refute* $\forall\overline{y}\psi(\overline{t}, \overline{y})$ *i.e.*, try to find a witness for $\exists\overline{y}\neg\psi(\overline{t}, \overline{y})$. If no witness is found, *i.e.*, if $\forall\overline{y}\psi(\overline{t}, \overline{y})$ is *validated* then output $\overline{t}$; otherwise backtrack to Stage 1 and find another witness for $\exists\overline{x}(\psi(\overline{x}, \overline{t_1}) \wedge \ldots \wedge \psi(\overline{x}, \overline{t_n}))$?

A completeness result would show that RichProlog's outputs stabilize to a correct witness $\overline{t}$ as soon as the finite fragment of T being dealt with is large enough, and as soon as enough sequences of terms $\overline{t_1}, \ldots, \overline{t_n}$ have been chosen during Stage 1. We think that the completeness results we have obtained so far can be extended to larger classes of generalized logic programs and Σ_2 queries, so we do not address the issue in this paper. Note that backtracking takes place withing Stage 1, within Stage 2, and from Stage 2 to Stage 1. RichProlog needs to validate the witness output every time Stage 1 is exited. It involves trying to refute a Π_1 sentence, hence trying to prove a Σ_1 sentence ξ. Note that negation has to occur either in the matrix of ξ or in the matrix of the initial Σ_2 query. A solution that works in many cases is to consider only Σ_2 queries in which the negated atoms are members of E only, and to use in Stage 2 a Σ_1 query ξ' that is logically equivalent to ξ modulo the background knowledge, but such that the negated atoms in ξ' are also members of E only. (As noticed above, the definition of a generalized clause and the notion of generalized logical consequence are such that negation applied to members of E can be trivially handled by negation as failure.) We will briefly return to the issue of negation in Section 3.4.

The search performed during Stage 1 is not very efficient. Indeed, instead of instantiating the universally quantified variables of the initial query by $\overline{t_i}$, $1 \leq i \leq n$, we could *target* $\overline{t_i}$, solving the query $\exists\overline{x}\exists\overline{y}\psi(\overline{x}, \overline{y})$? and making sure that the witness $(\overline{t}, \overline{t'})$ for this query is such that $\psi(\overline{t}, \overline{t'})$ is an instance of $\psi(\overline{t}, \overline{t_i})$ (equivalently, such that $\overline{t'}$ is an instance of $\overline{t_i}$). If $\overline{t'}$ contains variables, then some of the sequences of terms $\overline{t_{i+1}}, \ldots, \overline{t_n}$ could be instances of $\overline{t'}$ and 'skipped.' On the other hand, when it is known that Stage 1 is always exited with a witness $(\overline{t}, \overline{t'})$ for the query $\exists\overline{x}\exists\overline{y}\psi(\overline{x}, \overline{y})$? such that $\overline{t'}$ does not contain variables, the previous search strategy amounts to the one described above. The application to be presented in Section 4 has this property, hence we have sacrificed generality for simplicity and coherence. The reader interested in the more complex search strategy implemented by RichProlog during Stage 1 will find it described by the algorithm given in Section 3.4.

3.2 An Illustration

As described in the previous section, RichProlog proposes witnesses using a strategy that consists of two stages, where each stage is a search for solution to some Σ_1 query. The following illustration is presented with the purpose of exemplifying how a search tree for RichProlog is structured. It differs from the structure of a search tree for Prolog due to the confirmation process of Stage 2. The reader should note that the following illustration is not a search tree associated with an actual program execution; hence the assignments to variables are coherent and plausible, but arbitrary. Space constraints prevent us from presenting an actual example. We consider the example where the

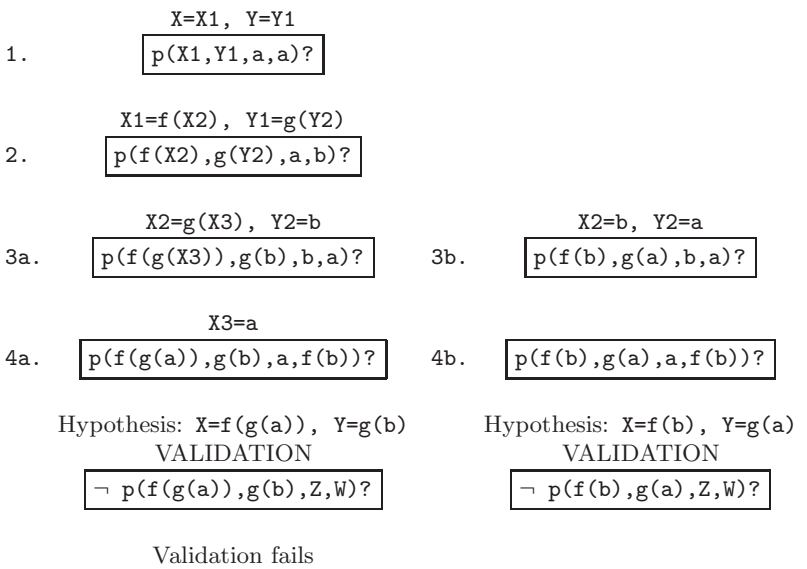


Fig. 1. An example of search tree for the query $\exists$X Y$\forall$Z W p(X, Y, Z, W)?

query is $\exists X \exists Y \forall Z \forall W\, p(X, Y, Z, W)$?. Assume that we decide exiting Stage 1 when a common witness for $\exists X \exists Y\, p(X, Y, a, a)$?, $\exists X \exists Y\, p(X, Y, a, b)$?, $\exists X \exists Y\, p(X, Y, b, a)$?, $\exists X \exists Y\, p(X, Y, a, f(b))$? is found. A possible search tree for this query is depicted in Figure 1. At the beginning of the search, the pair of variables `(Z,W)` is assigned the first pair of closed terms `(a,a)` (step 1.). The variables `Z` and `W` are subsequently assigned the values of the other pairs of closed terms in the following iterations: `(a,b)` at step 2., `(b,a)` at step 3., `(a,f(b))` at step 4. The instances of the existentially quantified variables become more and more specific since they have to cover more and more instances of the universally quantified variables: they become `(X1,Y1)` at step 1., `(f(X2),g(Y2))` at step 2., `(f(g(X3)),g(b))` at step 3a., and `(f(g(a)),g(b))` at step 4a. Following step 4a., enough instances of the universally quantified variables have been covered, and Stage 1 of the

search strategy is exited with a witness equal to `(f(g(a)),g(b))`. The execution now begins to validate $\forall \mathtt{Z}, \mathtt{W}\ \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{a})), \mathtt{g}(\mathtt{b}), \mathtt{Z}, \mathtt{W})$. This is done by trying to find a counterexample, *i.e.*, carrying out the search for a possible solution to the query $\exists \mathtt{Z}, \mathtt{W} \neg \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{a})), \mathtt{g}(\mathtt{b}), \mathtt{Z}, \mathtt{W})$?. Suppose that this search actually succeeds, *i.e.*: some witness for $\exists \mathtt{Z}, \mathtt{W} \neg \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{a})), \mathtt{g}(\mathtt{b}), \mathtt{Z}, \mathtt{W})$? is found. Then $(\mathtt{f}(\mathtt{g}(\mathtt{a})), \mathtt{g}(\mathtt{b}))$ is discarded and the search backtracks to step 4. If the search for a new witness for the query $\exists \mathtt{X3}\, \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{X3})), \mathtt{g}(\mathtt{b}), \mathtt{a}, \mathtt{b})$? fails, then the whole search backtracks to step 3. In Figure 2, it is assumed that a witness for $\exists \mathtt{X2}, \mathtt{Y2}\, \mathtt{p}(\mathtt{f}(\mathtt{X2}), \mathtt{g}(\mathtt{Y2}), \mathtt{b}, \mathtt{a})$? is found (step 3b.) that is also a witness for the next query (step 4b.). Enough instances of the universally quantified variables have been covered, and Stage 1 of the search strategy is exited with a witness equal to `(f(b),g(a))`. The execution now begins to validate $\forall \mathtt{Z}, \mathtt{W}\ \mathtt{p}(\mathtt{f}(\mathtt{b}), \mathtt{g}(\mathtt{a}), \mathtt{Z}, \mathtt{W})$. No counterexample is found and $(f(b), g(a))$ is output (possibly by stopping the unsuccessful search for a counterexample).

3.3 Complex Queries

In Section 3.1, we have considered Σ_2 queries whose matrix is an atomic formula. We now examine how to deal with more complex queries, transforming them into queries whose matrix is an atomic formula modulo an extension of T with a set of generalized clauses. Remember from the previous section how generalized clauses have been defined from the set denoted $\widehat{\mathcal{E}}$ and a set A of atomic formulas. Here we consider *generalized* Σ_2 *queries* defined as Σ_2 sentences whose matrix is built from $A \cup \widehat{\mathcal{E}}$ using disjunction and conjunction only. Basically, a new predicate symbol is introduced for each inner node in the parse tree of the matrix of the query. These symbols are not part of the vocabulary S; they are new symbols that only appear in a program derived from T—the initial generalized program—together with a particular generalized Σ_2 query. Each of these new predicate symbols must be of arity equal to the total number of (both the existentially quantified and the universally quantified) variables in the query. The query's matrix itself is replaced by the new predicate assigned to the root of the parse tree. Figure 2 depicts the parse tree for the matrix of the query $\varphi = \exists \mathtt{X} \exists \mathtt{Y} \forall \mathtt{Z} \psi$ where $\psi = \mathtt{p}_1(\mathtt{X}, \mathtt{Y}) \wedge [\neg \mathtt{p}_2(\mathtt{Z}) \vee (\mathtt{p}_3(\mathtt{X}, \mathtt{Y}) \wedge \mathtt{p}_4(\mathtt{Y}) \wedge (\neg \mathtt{p}_2(\mathtt{Y}) \vee \neg \mathtt{p}_3(\mathtt{X}, \mathtt{Z})))]$. Note that it is not necessary to introduce new predicate symbols for the leaves of the parse tree, and that the leaves can be labelled with atomic as well as with negations of atomic formulas which in the latter case, are necessarily negations of members of E. In this example, four new predicates have been created. Correspondingly, some generalized clauses will be created in the transformation of a particular generalized Σ_2 query, and will be used in the proof for that query only. In the following, 'the predicate symbol at node N' will denote the new predicate symbol introduced at N if N is an inner node, and the predicate symbol (from S) when N is a leaf. Basically, if an inner node is an *and-node*, then one generalized clause is created whose head is the predicate symbol at that node, and whose body is the conjunction of the predicate symbols at the children of the node. On the other hand, if the inner node is an *or-node*, then for each child of the node, one generalized clause is added to the program. The head of these generalized

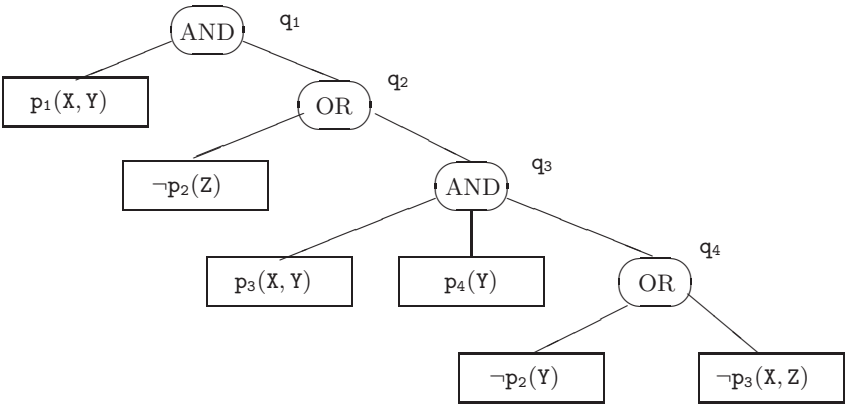


Fig. 2. Parse tree for $\mathtt{p}_1(\mathtt{X},\mathtt{Y})\wedge[\neg\mathtt{p}_2(\mathtt{Z})\vee(\mathtt{p}_3(\mathtt{X},\mathtt{Y})\wedge\mathtt{p}_4(\mathtt{Y})\wedge(\neg\mathtt{p}_2(\mathtt{Y})\vee\neg\mathtt{p}_3(\mathtt{X},\mathtt{Z})))]$

clauses is the predicate symbol at the parent node, while their body is the predicate symbol at the child node. In case of the previous example, the generalized clauses are given below and the query will be transformed into $\exists\mathtt{X}\exists\mathtt{Y}\forall\mathtt{Z}\ \mathtt{q}_1(\mathtt{X},\mathtt{Y},\mathtt{Z})$.

$$\mathtt{q}_1(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{p}_1(\mathtt{X},\mathtt{Y}),\mathtt{q}_2(\mathtt{X},\mathtt{Y},\mathtt{Z})$$
$$\mathtt{q}_2(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \neg\mathtt{p}_2(\mathtt{Z})$$
$$\mathtt{q}_2(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{q}_3(\mathtt{X},\mathtt{Y},\mathtt{Z})$$
$$\mathtt{q}_3(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{p}_3(\mathtt{X},\mathtt{Y}),\mathtt{p}_4(\mathtt{Y}),\mathtt{q}_4(\mathtt{X},\mathtt{Y},\mathtt{Z})$$
$$\mathtt{q}_4(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \neg\mathtt{p}_2(\mathtt{Y})$$
$$\mathtt{q}_4(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \neg\mathtt{p}_3(\mathtt{X},\mathtt{Z})$$

3.4 Further Remarks

As has been explained at the end of Section 3.1, RichProlog's search strategy for Stage 1 is more complex than the simple approach that has been described above. We now give, without comment for lack of space, the nondeterministic version of the algorithm used by RichProlog for Stage 1. Let μ denote a measure over the set of all closed terms, extended to a measure over the set of all terms by $\mu(t) = \mu(\{t' \mid t' \text{ is a closed instance of } t\})$. The m-product of μ is also denoted μ. We assume that some enumeration of all m-tuples of closed terms is given. Figure 3 describes the algorithm where δ represents a threshold that used together with μ, plays the role of the number n of atoms in the query of Stage 1. Remember that starting from a generalized Σ_2 query, the validation process requires trying to prove a Σ_1 query ξ, where, apart from trivial cases, negations of atomic sentences not in E occur. We have defined generalized logic programs and generalized Σ_2 queries so as to bypass the issue of true negation, but this issue seems to pop up inevitably at Stage 2 of the basic strategy. Still, in this paper and in most applications, we can avoid having to deal with true negation even for the validation process: it suffices that the formula $\exists\overline{y}\neg\psi(\overline{t},\overline{y})$ be logically equivalent

Input: A finite logic program T, a sentence of the form $\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m \psi$ where ψ is atomic, and a rational number δ in $(0, 1)$.
Output: A witness for $\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m \psi$.

Initialize Y to $\emptyset$.
Initialize $(t_1, \ldots, t_n)$ to $(x_1, \ldots, x_n)$.
While $\mu(Y) < \delta$ *do*

1. $Res = \{\psi\}$, $k = 0$.
2. $(t'_1, \ldots, t'_m)$ = the first member of the enumeration of m-tuples of closed terms that is not an instance of a member of Y.
3. *While* $Res \neq \emptyset$ *do*
   3.1. Choose ρ in Res and (renamed) clause $\alpha_0 \leftarrow \alpha_1 \ldots \alpha_p$ in T such that
   - ρ and α_0 unify with mgu θ_k;
   - $(t'_1, \ldots, t'_m)$ is an instance of $(y_1, \ldots, y_m)\theta_0 \ldots \theta_k$.

   (If no such ρ and $\alpha_0, \ldots, \alpha_p$ exist, exit 3.)
   3.2 Replace ρ by $\alpha_1, \ldots, \alpha_p$ in Res.
   3.3 Apply θ_k to Res.
   3.4 $k = k + 1$.
4. *If* $Res \neq \emptyset$ *then* output *no*.
5. *Else* let θ be a most general such that:
   - no variable occurs both in $(t_1, \ldots, t_n)\theta_0 \ldots \theta_k\theta$ and $(y_1, \ldots, y_m)\theta_0 \ldots \theta_k\theta$;
   - $(t'_1, \ldots, t'_m)$ is an instance of $(y_1, \ldots, y_m)\theta_0 \ldots \theta_k\theta$.
6. $(t_1, \ldots, t_n) = (t_1, \ldots, t_n)\theta_0 \ldots \theta_k\theta$.
7. $Y = Y \cup \{(y_1, \ldots, y_m)\theta_0 \ldots \theta_k\theta\}$.

Output $(t_1, \ldots, t_n)$.

Fig. 3. A nondeterministic algorithm for a more efficient search during Stage 1

to a Σ_1 query whose matrix is like the matrix of a generalized Σ_2 query: obtained from $A \cup \widehat{\mathcal{E}}$ using disjunction and conjunction only. This will be illustrated in the application to be examined now.

4 An Application

4.1 Description of the Problem

In this section, we show how RichProlog can be used to solve the problem of learning nonerasing pattern languages from positive data (see [1,2]). Consider the alphabet $\{\mathtt{a}, \mathtt{b}\}$ and an infinite sequence of variables $\mathtt{V1}, \mathtt{V2}, \ldots$. Define *words* as nonempty finite sequences over $\{\mathtt{a}, \mathtt{b}\}$, and *patterns* as nonempty finite sequences over $\{\mathtt{a}, \mathtt{b}, \mathtt{V1}, \mathtt{V2}, \ldots\}$. Hence words are particular kinds of patterns. Given a pattern π, an *instance* of π is a word obtained from π by replacing all variables in π by words, with the same replacement for occurrences of the same variables. A word w and a pattern π are said to *match* iff w is an instance of π. Consider a learner f which is presented with every finite initial segment of a (clearly infinite) enumeration of all instances of π, for some arbitrary pattern π. The task of f

is to discover π in the limit. The problem can be naturally cast in the logical framework as follows. Choose S such that constants and function symbols in S enable to represent a pattern by a closed term. Also put a predicate symbol P in S. A pattern π can then be represented by a Herbrand structure $\mathfrak{M}$ such that for all closed terms t, $\mathfrak{M} \models P(t)$ iff t represents a word that is an instance of π. Define the set E of possible evidence as $\{P(t) \mid \text{ closed term } t\}$. Remember that a possible theory T contains at least the E-diagram of a possible world $\mathfrak{M}$. In this example, the E-diagram of $\mathfrak{M}$ corresponds precisely to the set of words that are instances of the pattern represented by $\mathfrak{M}$. But T also contains a set X of possible axioms (which for RichProlog, must be a set of possible clauses) that are true in $\mathfrak{M}$. For this application, X can be defined independently of $\mathfrak{M}$, and will basically define the relationship between pattern and instances. Then we just have to ask a Σ_2 query whose intended meaning is: *does there exist a pattern π such that for all closed terms t that represent a word w, w is an instance of π iff $P(t)$ is observed*? This sentence will be a generalized logical consequence of $X \cup D_{\mathcal{E}}(\mathfrak{M})$ where $\mathfrak{M}$ is the structure that represents π. So in the limit, that is provided with enough possible evidence true in $\mathfrak{M}$, RichProlog outputs a witness t for the query such that t represents π.

4.2 Specifying the Problem in RichProlog

A *datum* will refer to an arbitrary word, a *positive datum* to some instance of the pattern to be learned, and a *negative datum* to other words.

The Hypothesis Space The learner will have access to the hypothesis space (the set of patterns) by the ability to generate every pattern. Patterns of length N will be generated before patterns of lenght $N+1$. The rules for this generation are 1.,2.,6.–13. in Figure 4. Basically, the predicate `pattern(Pi,N)` is true iff `Pi` is a pattern of length `N`. Starting from a general pattern (constant `gen_pattern)` of lenght `N`, equal to `V1V2...VN` up to a renaming of variables, a variable `Vi`, $1 \leq$ `i` $\leq$ `N`, can be replaced by `a` (thanks to the binary function symbol `subst_a`) or by `b` (thanks to the binary function symbol `subst_b`). Also given a variable `Vi`, $1 \leq$ `i` $\leq$ `N`, a variable `Vj` with $1 \leq$ `j` $<$ `i` can be picked up (using the binary predicate symbol `select`), and `Vi` and `Vj` can be identified (thanks to the ternary function symbol `eq_var`).

The Data The unary predicate symbol `p` is used to present positive data to the learner. Before asking a query, users should insert a number of statements corresponding to the data given to the learner. At any point, using negation as failure, the learner will consider any word which has not yet been presented (and maybe will never be) as not being an instance of the pattern to be learned. Obviously, some of these words will be presented later in time. For this application, this approach is not misleading, as will be explained below. For other applications involving learning from positive data only, negation as failure is not the right approach and true negation has to be dealt with in one way or another.

The Learner The learner must be able to determine whether a pattern and a word match or not. The binary predicate symbol `match` is to be interpreted as a relation between a pattern and a word, true when both match. The learner's strategy is to examine the shortest pattern which matches every positive datum, and does not match any other. It can be easily verified that a pattern of length `N` will not match at least one of the negative data of length `N`, if there is any such word. Due to the limiting nature of the problem, if a pattern `Pi` of length `N` is to be learned, then eventually all positive data of length `N` will be given to the learner. At that point, its assertion about the negative data of length `N` will be correct. Considering some words greater than `N` as negative data might be wrong, but it does not affect the learning strategy: the learner needs to validate hypotheses using *only* words of length `N`. For the implementation of `match` we refer to Figure 4. Essentially, `match` only succeeds when the pattern and the words are of the same length and match.

The Query We now consider the query that can be handled by RichProlog. The learner needs to hypothesize a pattern which matches all positive data, and does not match any word considered to be a negative datum. It does so by first generating possible hypotheses, and then trying to validate them. Generating a hypothesis can be done using the query:

$$\exists \mathtt{Pi}, \mathtt{N}[\mathtt{pattern}(\mathtt{Pi},\mathtt{N}) \wedge \forall \mathtt{W}((\mathtt{length}(\mathtt{W},\mathtt{N}) \wedge \mathtt{match}(\mathtt{Pi},\mathtt{W}) \rightarrow \mathtt{p}(\mathtt{W})) \wedge (\mathtt{length}(\mathtt{W},\mathtt{N}) \wedge \mathtt{mismatch}(\mathtt{Pi},\mathtt{W}) \rightarrow \neg \mathtt{p}(\mathtt{W})))]$$

It is easy to express this formula as a Σ_2 sentence whose matrix is built from $\widehat{\mathcal{E}}$ and the predicates `pattern`, `length`, `distinct`, `match` and `mismatch` using disjunction and conjunction only, hence to express this formula as a generalized Σ_2 query. Then we transform this generalized Σ_2 query into a Σ_2 sentence whose matrix is an atomic formula, and we add to the program the following set of generalized clauses, as explained in Section 3.3.

```
q(Pi,N,W)  :- pattern(Pi,N),q1(Pi,N,W).
q1(Pi,N,W) :- p(W),q2(Pi,N,W).
q1(Pi,N,W) :- not p(W),q3(Pi,N,W).
q2(Pi,N,W) :- length(W,N1),distinct(N1,N).
q2(Pi,N,W) :- match(Pi,W).
q3(Pi,N,W) :- length(W,N1),distinct(N1,N).
q3(Pi,N,W) :- mismatch(Pi,W).
```

Validating the hypothesis involves the following refuting sentence, which aims at finding counterexamples for the proposed witness:

$$\exists \mathtt{W}[\neg \mathtt{pattern}(\mathtt{Pi},\mathtt{N}) \vee (\mathtt{length}(\mathtt{W},\mathtt{N}) \wedge \mathtt{match}(\mathtt{Pi},\mathtt{W}) \wedge \neg \mathtt{p}(\mathtt{W})) \vee (\mathtt{length}(\mathtt{W},\mathtt{N}) \wedge \neg \mathtt{match}(\mathtt{Pi},\mathtt{W}) \wedge \mathtt{p}(\mathtt{W}))]$$

Note that in the validation process, the pair `(Pi,N)` is assigned a pair of closed terms, thus $\neg \mathtt{pattern}(\mathtt{Pi},\mathtt{N})$ will always be false. Hence it can be removed from

```
 1. numb(z).
 2. numb(s(N)) :- numb(N).
 3. distinct(s(N),z).
 4. distinct(z,s(N)).
 5. distinct(s(N),s(M)) :- distinct(N,M).
 6. select(N,s(N)).
 7. select(N,s(N)) :- select(N,N).
 8. pattern(Pi,s(N)) :-
        numb(N), pattern(Pi,s(N)).
 9. pattern(gen_pattern,z).
10. pattern(Pi,s(N)) :- pattern(Pi,N).
11. pattern(subst_a(Pi,s(N)),s(N)) :-
        pattern(Pi,N).
12. pattern(subst_b(Pi,s(N)),s(N)) :-
        pattern(Pi,N).
13. pattern(eq_var(Pi,s(M),s(N)),s(N)) :-
        select(M,N), pattern(Pi,N).
14. length(e,z).
15. length(a(W),s(N)) :- length(W,N).
16. length(b(W),s(N)) :- length(W,N).
17. symb_a(a(W),s(N)) :- length(W,N).
18. symb_a(a(W),N) :- symb_a(W,N).
19. symb_a(b(W),N) :- symb_a(W,N).
20. symb_b(b(W),s(N)) :- length(W,N).
21. symb_b(a(W),N) :- symb_b(W,N).
22. symb_b(b(W),N) :- symb_b(W,N).
23. match(gen_pattern,W).
24. match(subst_a(Pi,N),W) :-
        symb_a(W,N), match(Pi,W).
25. match(subst_b(Pi,N),W) :-
        symb_b(W,N), match(Pi,W).
26. match(eq_var(Pi,M,N),W) :-
        symb_a(W,N), symb_a(W,M),
            match(Pi,W).
27. match(eq_var(Pi,M,N),W) :-
        symb_b(W,N), symb_b(W,M),
            match(Pi,W).
28. mismatch(subst_a(Pi,N),W) :-
        symb_b(W,N).
29. mismatch(subst_b(Pi,N),W) :-
        symb_a(W,N).
30. mismatch(eq_var(Pi,M,N),W) :-
        symb_a(W,N), symb_b(W,M).
31. mismatch(eq_var(Pi,M,N),W) :-
        symb_b(W,N), symb_a(W,M).
32. mismatch(subst_a(Pi,N),W) :-
        mismatch(Pi,W).
33. mismatch(subst_b(Pi,N),W) :-
        mismatch(Pi,W).
34. mismatch(eq_var(Pi,N,M),W) :-
        mismatch(Pi,W).
```

Fig. 4. Logic program for learning pattern languages in the limit

the validation query. It is then easy to express the refuting sentence as a Σ_1 sentence whose matrix is built from $\widehat{\mathcal{E}}$ and the predicates `length`, `match` and `mismatch` using disjunction and conjunction only. We transform again this Σ_1 sentence into a Σ_1 sentence whose matrix is an atomic formula, and we add to the program the following set of generalized clauses, as explained in Section 3.3.

$$\texttt{r(Pi,N,W)} \;\texttt{:-}\; \texttt{p(W)}, \texttt{length(W,N)}, \texttt{mismatch(Pi,W)}.$$
$$\texttt{r(Pi,N,W)} \;\texttt{:-}\; \texttt{not p(W)}, \texttt{length(W,N)}, \texttt{match(Pi,W)}.$$

Remarks We illustrate the behavior in the limit of the learner who has to learn the pattern `V0V0V0`. First `aaa` is presented. So `a, b, aa, ab, ba, bb, bab, bbb, aba` are treated (by negation as failure) as negative data. On the basis of `aaa` alone, the learner outputs the hypothesis `V2aa`. This hypothesis will be refuted by the (supposedly) negative datum `baa`. So the search will backtrack to Stage 1, and `aaV0` will be the new witness. At some point, the learner is given the datum `bbb`. It will then output the hypothesis `V1V1V0`. Since `bba` does not belong to the list of positive data, it will be treated as a negative datum. The

learner will then output the correct hypothesis `VOVOVO`. As can be verified, as soon as the correct hypothesis has been output, the learner will keep it in the face of any new data. RichProlog has been run on this program, which did not raise any efficiency issues. The purpose was just to demonstrate the feasibility of the approach. We are working on larger applications in order to investigate in depth how well RichPolog can compete against alternative approaches.

5 Conclusion

Generalized logic, generalized logic programming, and RichProlog can be seen as instances of a natural generalization, or 'lifting,' of classical logic, logic programming, and Prolog. This enrichment is based on a notion of generalized logical consequences that are not necessarily compact, but account for deductive inferences, inductive inferences, and inferences of higher complexity. In this paper, we gave an overview of the main aspects of the three components of this work, instead of a full account of some part of it. In particular, we have not touched upon the issue of the class of generalized logic programs and queries for which RichProlog provides a complete proof procedure. It should be noted that the notion of completeness of classical logic is inappropriate here, since RichProlog is designed to perform noncompact inferences: completeness has to be based on the concept of convergence in the limit. Our main aim was to show that RichProlog extends the ideal of declarative programming to applications such as learning. We argued that learning problems can also share the benefit of declarative programming, despite the fact that learning is by nature nondeductive.

References

1. Angluin, D.: *Finding patterns common to a set of strings.* Journal of Computer and System Sciences. **21** (1980) 46–62 249
2. Angluin, D.: *Inductive Inference of Formal Languages from Positive Data.* Information and Control. **45** (1980) 117–135 249
3. Apt, K., Bol, R.: *Logic Programming and Negation: A Survey.* Journal of Logic Programming. **19/20** (1994) 177–190 239
4. Doets, K.: *From Logic to Logic Programming.* The MIT Press. (1994) 242
5. Jain, S., Osherson, D., Royer, J., Sharma, A. *Systems that learn: An Introduction to Learning Theory, Second Edition.* The MIT Press. (1999) 241
6. Kelly, K.: *The Logic of Reliable Inquiry.* Oxford University Press. (1996) 241
7. Le, T.: *A general scheme for representing negative and quantified queries for deductive databases.* Proceedings of the First Internatinal Conference on Information and Knowledge Management. Baltimore, Maryland. (1992) 239
8. Lifschitz, V.: *Closed-World Databases and Circumscription.* Artificial Intelligence. **27** (1985) 229–235 242
9. Martin, E., Osherson, D.: *Elements of Scientific Inquiry.* The MIT Press. (1998) 241
10. Martin, E., Sharma, A., Stephan, F.: *A General Theory of Deduction, Induction, and Learning.* In Jantke, K., Shinohara, A.: Proceedings of the Fourth International Conference on Discovery Science. Springer-Verlag. (2001) 228–242 242, 243

11. Reiter, R.: *On Closed-World Data Bases*. In Gallaire, J., Minker, J., ed., Logic and Data Bases. Plenum Press. (1978) 55–76 242
12. Shepherdson, J.: *Negation in Logic Programming*. In Minker, J., ed., Foundations of Deductive databases and Logic Programming. Morgan Kaufmann. (1988) 19–88 239
13. Shoham, Y.: *Reasoning about change*. The MIT Press. (1988) 242

Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification

François Bry and Sebastian Schaffert

Institute for Computer Science, University of Munich
http://www.pms.informatik.uni-muenchen.de

Abstract. The growing importance of XML as a data interchange standard demands languages for data querying and transformation. Since the mid 90es, several such languages have been proposed that are inspired from functional languages (such as XSLT [1]) and/or database query languages (such as XQuery [2]). This paper addresses applying logic programming concepts and techniques to designing a declarative, rule-based query and transformation language for XML and semistructured data. The paper first introduces issues specific to XML and semistructured data such as the necessity of flexible "query terms" and of "construct terms". Then, it is argued that logic programming concepts are particularly appropriate for a declarative query and transformation language for XML and semistructured data. Finally, a new form of unification, called "simulation unification", is proposed for answering "query terms", and it is illustrated on examples.

1 Introduction

This article addresses applying logic programming to XML and semistructured data querying and transformation. This issue is of growing importance in both, practice and research. XML now is the data interchange standard of choice in application areas such as e-commerce, molecular biology, and astronomy. Furthermore, with the XML application XHTML, XML is becoming the preferred standard for textual web contents. Also recent languages for static and/or animated graphics such as SVG (Scalable Vector Graphics), X3D (an XML-based redefinition of the Virtual Reality Modelling Language VRML), and SMIL (Synchronised Multimedia Integration Language) are XML applications. "Native XML" database management systems are already marketed.

As a consequence, the design and implementation of selector and query languages for XML such as CSS selectors and XPath are premier concerns of the World Wide Web Consortium (W3C). Selector languages such as CSS selectors and XPath have been developed initially for style-sheet and/or transformation languages such as CSS and XSLT. Selector languages are "path-oriented", i.e. a node (i.e. a subterm) in a tree (i.e. a term) is specified in terms of a root-to-node path in the manner of the file selection formalisms of operating systems. Constructs from regular expressions languages such as $*$, $+$, $|$, ? and "wildcards" give

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 255–270, 2002.
© Springer-Verlag Berlin Heidelberg 2002

rise to expressing node repetitions, options, and nodes with unspecified labels. These constructs are essential in selecting and/or querying XML and semistructured data, for a basic principle of XML and semistructured data is that data items, called "documents", do not have to conform to a predefined schema (expressed in XML in the DTD or XML Schema formalisms). This principle aims at facilitating the interchange of data in unnormed contexts like the World Wide Web. XML-like tree structured data items that do not necessarily conform to a schema are called "semistructured" in database research [3].

Query languages [4] inspired from SQL and OQL [5] have been developed for XML and semistructured data. Queries in these languages for XML and semistructured data in general consist of two parts: a query proper and a construct part. The query part expresses a selection of nodes (i.e. subterms) from a data item (i.e. term). Node selection is in general expressed in a path-oriented formalism à la XPath extended with "and" and "or" connectives. The construct parts serve to re-assemble into new terms (some of) the nodes (i.e. terms) selected in the query part. The construct part is also called restructuring or transformation part. For example, the following query from [6] (there with `books-with-prices` instead of `book-price-comparison`) expresses in XML Query [2], also called XQuery, the request to list for each book found as `entry` element at `amazon.com` and as `book` element at `bn.com` the book's title and the prices at both sources:

```
<book-price-comparison>
   { for $a in document("www.amazon.com/reviews.xml")//entry,
         $b in document("www.bn.com/bib.xml")//book
     where $b/title = $a/title
     return
        <book-with-prices>
           { $b/title }
           <price-amazon>{  $a/price/text() } </price-amazon>
           <price-bn>{  $b/price/text() } </price-bn>
        </book-with-prices>
    }
</book-price-comparison>
```

The query part is contained between the first `{` and `return`. The construct part is specified in the `book-with-prices` element. Note the node selection (expressed with the XPath expressions `$a/price/text()` and `$b/price/text()`) contained in the construct part.

The work reported about in this paper is based upon the conviction that logic programming provides with concepts giving rise to query and transformation languages more declarative than those based on a path-oriented node selection. A query term inspired from a Prolog goal atom would give rise to a "context-conscious" selection of several nodes (i.e. subterms) within a same term (the "context") at a time, a term inspired from a Prolog head atom would be a convenient construct expression, rules relating (conjunctions or disjunctions of) query terms to construct terms would define views (in the database sense) and

give rise to a deduction-like (backward or forward) chaining of term constructions – a feature often needed in XML and semistructured data processing.

In such a language inspired from logic programming, the previous query example can be expressed as follows (symbols beginning with upper case latters denote variables):

```
construct
   <book-price-comparison>
      all <book-with-prices>
            <title> T</title>
            <price-amazon> Pa</price-amazon>
            <price-bn> Pb</price-bn>
          </book-with-prices>
   </book-price-comparison>
where
   in amazon.com:
   <entry>
      <title>T<title>
      <price>Pa</price>
   </entry>
   and
   in bn.com:
   <book>
      <title>T</title>
      <price>Pb</price>
   </book>
```

In a more conventional syntax, this query can be expressed as follows (an element name is shortened to the first letters of its constituting words and the locations `amazon.com` and `bn.com` are omitted):

```
bpc{ all bwp[t[T],pa[Pa],pb[Pb]] } ← e[t[T],p[Pa]] and b[t[T],p[Pb]]
```

An advantage of such a rule is to clearly separate node selection, expressed only in the query terms i.e. in the rule body, from construction, expressed in the construct term i.e. in the rule head. This is beneficial for both, the programmer and query evaluation. Another advantage of the approach is to avoid the rather procedural navigation through data item imposed by a path-oriented node selection. In the rule given above, the contents of both elements `t` (i.e. `title`) and `pa` (i.e. `price-amazon`) are selected in a single query term `e[t[T], p[Pa]]`. In contrast, the XQuery expression needs two paths for the same selection, `$a/title` and `$a/price/`. The query term `e[t[T], p[Pa]]` stresses the common context and the relative position of the selected nodes (i.e. subterms) `T` and `Pa`. In contrast, the paths `$a/title` and `$a/price/` specify two independent navigations through a term. Arguably, a term-oriented (or context-conscious or positional) node selection is more declarative than a path-oriented (or navigational) node selection.

This paper reports about first achievements in designing a term-oriented, "context-conscious", or "positional" query and transformation language for XML and semistructured data. In order to conform to the semistructured data

paradigm, a novel form of unification is needed. This paper is mostly devoted to motivating and specifying a nonstandard unification, called "simulation unification" convenient for a positional querying and transformation of XML and semistructured data.

This article is organised as follows. Section 1 is this introduction. Section 2 describes those aspects of the query and transformation language under development that are relevant to this paper. Simulation unification is addressed in Section 3. Section 4 is devoted to related work and a conclusion.

2 Elements of a Query and Transformation Language

This section introduces into those aspects of an experimental query and transformation language for XML and semistructured data, called Xcerpt, that are relevant to this paper. Aspects of XML, such as attributes and namespaces, that are irrelevant to this paper, are not explicitly addressed in the following. Two disjoint sets of symbols, the set $\mathcal{L}$ of labels (or tags) and the set $\mathcal{V}$ of variables are considered. Labels (variables, resp.) are denoted by words starting with a lower (upper, resp.) case letter. The following meta-variables (with or without indices and/or superscripts) are used:

- l denotes a label,
- X denotes a variable,
- t denotes a term (as defined below).

2.1 Database Terms

Database terms are an abstraction of XML documents. Following a common practice in XML query language and semistructured data research [3], a database is a set (or multiset) of database terms and the children of a document node may be either ordered (as in SGML and in standard XML), or unordered (as in the semistructured data model). In the following, a term whose root is labelled l and has *ordered* children $t_1, \ldots, t_n$ is denoted $l[t_1, \ldots, t_n]$; a term whose root is labelled l and has *unordered* children $t_1, \ldots, t_n$ is denoted $l\{t_1, \ldots, t_n\}$.

Definition 1 (Database Terms). *Database terms are inductively defined as follows:*

1. *A label is a (atomic) database term.*
2. *If l is a label and $t_1, \ldots, t_n$ are $n \geq 1$ database terms, then $l[t_1, \ldots, t_n]$ and $l\{t_1, \ldots, t_n\}$ are database terms.*

Database terms are similar to classical logic ground terms except that, (1) the arity of a function symbol, called here "label", is not fixed (as in Prolog), and (2) the arguments of a function symbol may be unordered.

Whatever storage is used, a database term $t_0 = l\{t_1, \ldots, t_n\}$ with unordered subterms $t_1, \ldots, t_n$ will always be stored in a manner inducing an order on

$t_1, \ldots, t_n$. The notion of unordered subterms $t_1, \ldots, t_n$ means that (1) the storage ordering of $t_1, \ldots, t_n$ is left at the discretion of the storage system (giving rise e.g. to clustering as many t_i as possible on a secondary memory page), and (2) no given ordering is to be returned when t_0 is accessed.

In the following, $\mathcal{T}_{db}$ denotes the set of all database terms.

2.2 Query Terms

A query term is a "pattern" that specifies a selection of database terms very much like Prolog goal atoms and SQL selections. However, answers to query terms (cf. below Definition 13) differ from answers to Prolog goal atoms and SQL selections as follows:

- Database terms with additional subterms to those explicitly mentioned in a query term might be answers to this query term.
- Database terms with a different subterm ordering from that of the query term might be answers to this query term.
- A query term might specify subterms at an arbitrary depth.

In query terms, the single square and curly brackets, [ ] and { }, denote "exact subterm patterns", i.e. single (square or curly) brackets are used in a query term to be answered by database terms with no more subterms than those given in the query term. Double square and curly brackets, [[ ]] and {{ }}, on the other hand, denote "partial subterm patterns" as described above.

[ ] and [[ ]] are used if the subterm order in the answers is to be that of the query term, { } and {{ }} are used otherwise. Thus, possible answers to the query term $t_1 = a[b, c\{\{d, e\}\}, f]$ are the database terms $a[b, c\{d, e, g\}, f]$ and $a[b, c\{d, e, g\}, f\{g, h\}]$ and $a[b, c\{d, e\{g, h\}, g\}, f\{g, h\}]$ and $a[b, c[d, e], f]$. In contrast, $a[b, c\{d, e\}, f, g]$ and $a\{b, c\{d, e\}, f\}$ are no answers to t_1. The only answers to $f\{\ \}$ are f-labelled database terms with no children.

The construct *descendant*, short *desc*, introduces a subterm at an unspecified depth. Thus, possible answers to the query term $t_2 = a[desc\ f[c, d], b]$ are $a[f[c, d], b]$ and $a[g[f[c, d]], b]$ and $a[g[f[c, d], h], b]$ and $a[g[g[f[c, d]]], b]$ and $a[g[g[f[c, d], h], i], b]$.

In a query term, a variable X can be restricted to some query terms using the construct $\rightsquigarrow$, read "as". Thus, the query term $t_3 = a[X_1 \rightsquigarrow b[[c, d]], X_2, e]$ constrains the variable X_1 to such database terms that are possible answers to the query term $b[[c, d]]$. Note that the variable X_2 is unconstrained in t_3. Possible answers to t_3 are e.g. $a[b[c, d], f, e]$ which binds X_1 to $b[c, d]$ and X_2 to f, $a[b[c, d], f[g, h], e]$ which binds X_1 to $b[c, d]$ and X_2 to $f[g, h]$, $a[b[c, d, e], f, e]$ which binds X_1 to $b[c, d, e]$ and X_2 to f, and $a[b[c, e, d], f, e]$ which binds X_1 to $b[c, e, d]$ and X_2 to f.

Definition 2 (Query Terms). *Query terms are inductively defined as follows:*

1. *If l is a label, then l and $l\{\}$ are (atomic) query terms.*
2. *A variable X is a query term.*

3. *If X is a variable and t a query term, then $X \rightsquigarrow t$ is a query term.*
4. *If X is a variable and t is a query term, then $X \rightsquigarrow$ desc t is a query term.*
5. *If l is a label and $t_1, \ldots, t_n$ are $n \geq 1$ query terms, then $l[t_1, \ldots, t_n]$, $l\{t_1, \ldots, t_n\}$, $l[[t_1, \ldots, t_n]]$, and $l\{\{t_1, \ldots, t_n\}\}$ are query terms.*

Multiple variable constraints are not precluded. A possible answer to e.g. $a\{\{X \rightsquigarrow b\{\{c\}\}, X \rightsquigarrow b\{\{d\}\}\ \}\}$ is $a\{b\{c, d\}\}$. The query term $a[[X \rightsquigarrow b\{\{c\}\}, X \rightsquigarrow f\{\{d\}\}]]$, however, has no answers, as the labels b and f are distinct.

Subterms (of query terms) are defined as usual (e.g. a and X and $Y \rightsquigarrow$ *desc* $b\{X\}$ and $h\{a, X \rightsquigarrow k\{c\}\}$ and $X \rightsquigarrow k\{c\}$ and t itself are subterms of $t = f\{a, g\{Y \rightsquigarrow$ *desc* $b\{X\}, h\{a, X \rightsquigarrow k\{c\}\}\}\}$). In the following, query terms are assumed to be variable well-formed, a notion defined as follows.

Definition 3 (Variable Well-Formed Query Terms). *A term variable X* depends *on a term variable Y in a query term t if $X \rightsquigarrow t_1$ is a subterm of t and Y is a subterm of t_1. A query term t is* variable well-formed *if t contains no term variables $X_0, \ldots, X_n$ $(n \geq 1)$ such that 1. $X_0 = X_n$ and 2. for all $i = 1, \ldots, n$, X_i depends on X_{i-1} in t.*

E.g. $f\{X \rightsquigarrow g\{X\}\}$ and $f\{X \rightsquigarrow g\{Y\}, Y \rightsquigarrow h\{X\}\}$ are not variable well-formed. Variable well-formedness precludes queries specifying infinite answers. Usually terms that are not variable well-formed are called cyclic. However, Xcerpt also allows for arbitrary graph structures (which are not discussed in this paper, cf. [7]) which might by cyclic in another sense.

In the following, query terms are implicitly assumed to be variable well-formed and the set $\mathcal{T}_q$ is defined as the set of all (variable well-formed) query terms.

2.3 Construct Terms

Construct terms serve to re-assemble variables, the "values" of which are specified in query terms, so as to form new database terms. Thus, construct terms may contain both constructs [ ] and { } (like database terms) as well as variables. However, the construct $\rightsquigarrow$ is not allowed in construct terms, as variables should be constrained where they are defined, (i.e. in query terms), not in construct terms where they are used to specify new terms.

Definition 4 (Construct Terms). *Construct terms are inductively defined as follows:*

1. *A label l is a (atomic) construct term.*
2. *A variable X is a construct term.*
3. *If l is a label and $t_1, \ldots, t_n$ are $n \geq 1$ construct terms, then $l[t_1, \ldots, t_n]$ and $l\{t_1, \ldots, t_n\}$ are construct terms.*

The set of construct terms will be denoted with $\mathcal{T}_c$ in the rest of this paper. Note that $\mathcal{T}_{db} \subseteq \mathcal{T}_c \subseteq \mathcal{T}_q$.

2.4 Construct-Query Rules

Construct-query rules, short rules, relate queries, consisting of a conjunction of query terms, and construct terms. It is assumed (cf. below Point 3 of Definition 5) that each variable occurring in the construct term of a construct-query rule also occurs in at least one of the query terms of the rule, i.e. variables in construct-query rules are assumed to be "range-restricted" or "allowed". A relaxation of this condition like in Prolog does not seem to be desirable.

Definition 5 (Construct-Query Rule). *A construct-query rule is an expression of the form* $t^c \leftarrow t_1^q \wedge \ldots \wedge t_n^q$ *such that:*

1. $n \geq 1$ *and for all* $i = 1, \ldots n$, t_i^q *is a query term,*
2. t^c *is a construct term, and*
3. *every variable occurring in* t^c *also occurs in at least one of the* t_i^q.

The left hand-side, i.e. the construct term, of a (construct-query) rule will be referred to as the rule "head". The right hand-side of a (construct-query) rule will be referred to as the rule "body". Note that, in contrast to the body of a Prolog clause, the body of a (construct-query) rule cannot be empty, for empty rule bodies do not seem to be needed for the applications considered.

An Xcerpt program consists of a finite set of (construct-query) rules with a (conjunction of) query term(s). The scope of an occurrence of a variable in an Xcerpt program is, like in Prolog, restricted to the rule it occurs in.

2.5 Further Features

The full version of this paper [8] describes in more details further features of the experimental language Xcerpt, among others the construct *all* mentioned in the introduction.

3 Simulation Unification

The rule-based language Xcerpt, the main elements of which have been introduced above in Section 2, can be processed by both forward and backward chaining. Techniques similar to those used in implementations of Prolog (e.g. the use of the run-time stack for implementing a depth-first search) or of Datalog (e.g. a database storage of goal atoms) can be used for Xcerpt as well. However, Xcerpt cannot rely on standard unification because of the requirements on query terms listed in Section 2.2: A query term of the form $l[[t_1, \ldots, t_n]]$ or $l\{\{t_1, \ldots, t_n\}\}$ should "unify" with l-labelled terms with more subterms than those matching $t_1, \ldots,$ and t_n; also unordered subterms (like in $l\{\{t_1, \ldots, t_n\}\}$), the descendant construct *desc* and the *as* construct $\rightsquigarrow$ have to be dealt with. This section is devoted to introducing a nonstandard unification called "simulation unification" fulfilling these requirements.

For space reasons, simulation unification is defined in this paper under the assumptions that {{ }} and { } are the only kinds of braces, and that braces are

only allowed immediately on the right of a label (like in $f\{\{a, g\{b, c\}, d\}\}$) and not directly within other braces (like in $f\{\{a, \{b, c\}, d\}\}$). The full article [8] explains how to skip these restrictions.

3.1 Simulation

Intuitively, a simulation of a graph G_1 in a graph G_2 is a mapping of the nodes of G_1 in the nodes of G_2 preserving the edges. In other words, there exists a simulation of G_1 in G_2, if the node/edge structure of G_1 can be found as a subgraph of G_2. Efficient algorithms for computing simulation (bisimulation, resp.) are given e.g. in [9]. In [3,10], simulation is used for verifying the conformity of semistructured data to a schema. The language UnQL [11] introduces (bi)simulation for query answering, but the usage is restricted to pattern matching.

Definition 6 (Graph Simulation). *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs and let $\sim$ be an equivalence relation on $V_1 \cup V_2$. A relation $\mathcal{S} \subseteq V_1 \times V_2$ is a simulation with respect to $\sim$ of G_1 in G_2 if:*

1. *If $v_1\ \mathcal{S}\ v_2$, then $v_1 \sim v_2$.*
2. *If $v_1\ \mathcal{S}\ v_2$ and $(v_1, v_1') \in E_1$, then there exists $v_2' \in V_2$ such that $v_1'\ \mathcal{S}\ v_2'$ and $(v_2, v_2') \in E_2$.*

A simulation $\mathcal{S}$ of a tree T_1 with root r_1 in a tree T_2 with root r_2 is a rooted simulation of T_1 in T_2 if $r_1\ \mathcal{S}\ r_2$.

Note that the definition of a simulation $\mathcal{S}$ of G_1 in G_2 does not preclude that two distinct vertices v_1 and v_1' of G_1 are simulated by the same vertice v_2 of G_2, i.e. $(v_1, v_2) \in \mathcal{S}$ and $(v_1', v_2) \in \mathcal{S}$. Figure 1 gives examples of simulations (represented by the dashed edges) with respect to vertice label equality.

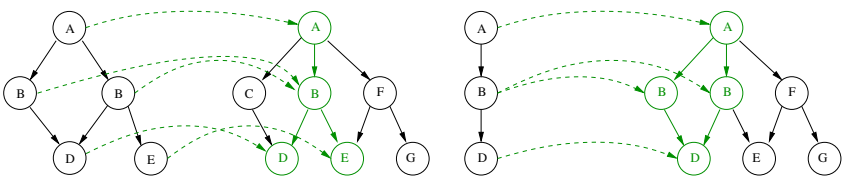


Fig. 1. Rooted Simulations (with respect to label equality)

Simulation with respect to label equality is a first notion towards a formalisation of answers to query terms: If a database term t^{db} is to be an answer to a query term t^q (both terms being considered as trees), then there must exist a rooted simulation with respect to label equality of (the term/tree with no $\rightsquigarrow$ and *desc* constructs subjacent to) t^q in t^{db}.

3.2 Term Lattice

Definition 7 (Ground Query Term). *A query term is* ground *if it contains no variables, no* $\rightsquigarrow$ *and no desc.*

In the following, the set of all ground query terms, extended by the two special terms $\bot$ (the "empty" term) and $\top$ (the "full" term) will be denoted by $\mathcal{T}_{\text{ground}}$. Note that $\mathcal{T}_{\text{ground}} \not= \mathcal{T}_{db}$, since in contrast to database terms ground query terms may contain both constructs { } and {{ }}.

Definition 8 (Ground Query Term Simulation). *Let* $t_1 \in \mathcal{T}_{ground}$ *and* $t_2 \in \mathcal{T}_{ground}$ *. Let* $S_i \subseteq \mathcal{T}_{ground}$ *denote the set of subtrees of* t_i *(*$i \in \{1, 2\}$*). A relation* $\mathcal{S} \subseteq S_1 \times S_2$ *is a simulation of* t_1 *in* t_2 *if:*

1. $t_1 \; \mathcal{S} \; t_2$
2. *If* $l_1 \; \mathcal{S} \; l_2$ *then* $l_1 = l_2$.
3. *If* $l_1\{\{t_1^1, \ldots, t_n^1\}\} \; \mathcal{S} \; l_2\{\{t_1^2, \ldots, t_m^2\}\}$*), then* $l_1 = l_2$ *and for all* $i \in \{1, \ldots, n\}$ *there exists* $j \in \{1, \ldots, m\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$
4. *If* $l_1\{\{t_1^1, \ldots, t_n^1\}\} \; \mathcal{S} \; l_2\{t_1^2, \ldots, t_m^2\}\}$*), then* $l_1 = l_2$ *and for all* $i \in \{1, \ldots, n\}$ *there exists* $j \in \{1, \ldots, m\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$*)*
5. *If* $l_1\{t_1^1, \ldots, t_n^1\} \; \mathcal{S} \; l_2\{\{t_1^2, \ldots, t_m^2\}\}$*), then* $l_1 = l_2$ *and for all* $i \in \{1, \ldots, n\}$ *there exists* $j \in \{1, \ldots, m\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$*), and for all* $j \in \{1, \ldots, m\}$ *there exists* $i\{1, \ldots, n\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$
6. *If* $l_1\{t_1^1, \ldots, t_n^1\} \; \mathcal{S} \; l_2\{t_1^2, \ldots, t_m^2\}$*), then* $l_1 = l_2$ *and for all* $i \in \{1, \ldots, n\}$ *there exists* $j \in \{1, \ldots, m\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$*, and for all* $j \in \{1, \ldots, m\}$ *there exists* $i\{1, \ldots, n\}$ *such that* $t_i^1 \; \mathcal{S} \; t_j^2$

Definition 9 (Simulation Preorder). $\preceq$ *is the preorder on* $\mathcal{T}_{ground} \setminus \{\bot, \top\}$ *defined by* $t_1 \preceq t_2$ *if there exists a ground query term simulation of* t_1 *in* t_2*.*

The preorder $\preceq$ is not an order, for although $t_1 = f\{a\} \preceq t_2 = f\{a, a\}$ and $t_2 = f\{a, a\} \preceq t_1 = f\{a\}$ (both a of t_2 can be simulated by the same a of t_1), $t_1 = f\{a\} \not= t_2 = f\{a, a\}$.

However, $\preceq$ induces as follows a (partial) order on $\mathcal{T}_{\text{ground}}$. First, consider the equivalence relation $\equiv$ on $\mathcal{T}_{\text{ground}}$ defined by the bisimulation $t_1 \equiv t_2$ if both, $t_1 \preceq t_2$ and $t_2 \preceq t_1$ hold. Since $\preceq$ is reflexive and transitive, $\equiv$ is also reflexive and transitive. $\equiv$ is by definition symmetric.

It is natural to chose as representative of an equivalence class of $\mathcal{T}_{\text{ground}} / \equiv$ the class member with the minimal number of repeated subterms, e.g. $f\{a\}$ is chosen as representative of class $\{f\{a\}, f\{a, a\}, f\{a, a, a\}, f\{a, a, a, a\}, \ldots\} \in \mathcal{T}_{\text{ground}} / \equiv$.

In the following, referring to this representative will always be meant as a reference to the whole equivalence class and the (partial) order induced by $\preceq$ on $\mathcal{T}_{\text{ground}} / \equiv$ will be noted $\preceq$, too. In other words, answers to query terms will be defined up to $\equiv$ as representatives of elements of $\mathcal{T}_{\text{ground}} / \equiv$. Intuitively, $t_1 \preceq t_2$ means that it is possible to remove from t_2 subterms at arbitrary depth, until the remaining term is either t_1 or some $\preceq$-smaller term from the same $\equiv$-class as t_1.

Definition 10 (Ground Query Term Lattice). *$\preceq$ is extended to $\bot$ and $\top$ as follows: For all $t \in \mathcal{T}_{ground}$, $\bot \preceq t$ and $t \preceq \top$. $(\mathcal{T}_{ground} / \equiv, \preceq)$ is the ground query term lattice.*

3.3 Answers

An answer in a database $D \subseteq \mathcal{T}_{db}$ to a query term t^q is characterised by a set of values for the variables in t^q such that the ground query term t^q_g resulting from substituting these values for the variables in t^q is simulated by an element t of D (i.e. $t^q_g \preceq t$).

Consider for example the query $t^q = f\{\{X \rightsquigarrow g\{\{b\}\}, X \rightsquigarrow g\{\{c\}\}\}\}$ against the database $D = \{f\{g\{a,b,c\}, g\{a,b,c\}, h\},\ f\{g\{b\}, g\{c\}\}\}$. The $\rightsquigarrow$ constructs in t^q yield the constraint $g\{\{b\}\} \preceq X \wedge g\{\{c\}\} \preceq X$. The first database term in D yields the constraint $X \preceq g\{a,b,c\}$. The second database term in D yields the constraint $X \preceq g\{b\} \wedge X \preceq g\{c\}$. The constraint $g\{\{b\}\} \preceq X \wedge g\{\{c\}\} \preceq X$ is incompatible with $X \preceq g\{b\} \wedge X \preceq g\{c\}$. Thus, the only possible value for X is $g\{a,b,c\}$ and the only possible answer to t^q in D is $t^q_a = f\{g\{a,b,c\}, g\{a,b,c\}, h\}$.

Note that, in contrast to Prolog and SQL, the binding $X = g\{a,b,c\}$ does not suffice to characterise the answer t^q_a, for t^q does not have any "handle" for the subterm h of t^q_a. If not only the bindings for X but the complete answers to t^q are sought for, then the query term $Y \rightsquigarrow f\{\{X \rightsquigarrow g\{\{b\}\}, X \rightsquigarrow g\{\{c\}\}\}\}$ is to be used instead of t^q.

Definition 11 (Substitutions and Instances). *Let t^q be a query term and let $X_1, \ldots, X_n$ be the variables occurring (left or right of $\rightsquigarrow$ or elsewhere) in t^q.*

A substitution is a function which assigns a construct term to each variable of a finite set of variables. A substitution σ is a grounding substitution for a query term t^q if σ assigns a ground query term to each variable in t^q.

If σ is a substitution (grounding substitution, resp.) for t^q assigning t_i to X_i $(1 \leq i \leq n)$, then the instances (ground instances, resp.) of t^q with respect to σ are those construct terms (ground query terms, resp.) that can be constructed from t^q as follows:

1. *Replace each subterm $X \rightsquigarrow t$ by X.*
2. *Replace each occurrence of X_i by t_i $(1 \leq i \leq n)$.*

Requiring in Definition 2 *desc* to occur to the right of $\rightsquigarrow$ makes it possible to characterise ground instances of query terms by substitutions. This is helpful for formalising answers but not necessary for language implementions.

Not all ground instances of a query term are acceptable answers, for some instances might violate the conditions expressed by the $\rightsquigarrow$ and *desc* constructs.

Definition 12 (Allowed Instances). *The constraint induced by a query term t^q and a substitution σ is the conjunction of all inequations $t\sigma \preceq X\sigma$ such that $X \rightsquigarrow t$ with $t \neq desc\ t_1$ is a subterm of t^q, and of all expressions $X\sigma \lhd t\sigma$ (read "$X\sigma$ subterm of $t\sigma$") such that $X \rightsquigarrow desc\ t$ is a subterm of t^q, if t^q has such*

subterms. If t^q has no such subterms, the constraint induced t^q and σ is the formula true.

Let σ be a grounding substitution of a query term t^q. The instance $t\sigma$ of t^q is allowed if:

- *$t\sigma \not= \bot$ and $t\sigma \not= \top$.*
- *Each inequality $t_1 \preceq t_2$ in the constraint induced by t^q and σ is satisfied in $(\mathcal{T}/\equiv, \preceq)$.*
- *If $t_1 \lhd t_2$ occurs in the constraint induced by t^q and σ, then there exists a subterm t_1' of t_1 such that $t_2 \preceq t_1'$*

Definition 13 (Answers). *Let t^q be a query term, D a database (i.e. $D \subseteq \mathcal{T}_{db}$). An answer to t^q in D is a database term $t^{db} \in D$ such that there exists an allowed instance t_a^q of t^q satisfying $t_a^q \preceq t^{db}$.*

3.4 Simulation Unification

Simulation unification is a non-deterministic method for solving inequations of the form $t^q \preceq t^c$, where t^q is a query term, t^c is a construct term (possibly a database term), and t^q and t^c are variable disjoint, in the database term lattice $(\mathcal{T}_{db}/\equiv, \preceq)$, i.e. to determine substitutions σ such that $t^q\sigma$ and $t^c\sigma$ have instances $t^q\sigma\tau$ and $t^c\sigma\tau$ such that $t^q\sigma\tau$ and $t^c\sigma\tau$ are database terms and $t^q\sigma\tau \preceq t^c\sigma\tau$ holds.

Such inequations may result from a forward chaining evaluation of a query-construct rule against database terms. In such a case, the right-hand side t^c of the inequation contains no variables, i.e. it is a database term. An inequation $t^q \preceq t^c$ may also result from a backward chaining evaluation of the query term t^q against a query-construct rule whose head is t^c. In such a case, variables may occur in the construct term t^c but t^q and t^c are variable disjoint. That t^q and t^c do not share variables follows from the variable scoping rule for Xcerpt programs postulated in Section 2.4 above (this is the so-called "standardisation apart" of deduction methods).

Simulation unification consists in repeated applications of Term Decomposition phases followed by a Consistency Verification phase to a formula C (for constraint store) consisting in disjunctions of conjunctions of inequations of the form $t^q \preceq t^c$ (with t^q query term and t^c construct term) and/or equations of the form $t_1^c = t_2^c$ (with t_1^c and t_2^c construct terms). At the beginning C consists in a single inequation $t^q \preceq t^c$. Both phases Term Decomposition and Consistency Verification consist in stepwise changes of the constraint store C. These changes are expressed in the following formalism inspired from [12]:

A "simplification" $L \Leftrightarrow R$ replaces L by R.

Trivially satisfied inequations or equations are replaced by the atomic formula *true.* Inconsistent conjunctions of inequations or equations are replaced by the atomic formula *false.*

Definition 14 (Term Decomposition Rules). *Let l (with or without indices) denote a label. Let t^1 and t^2 (with or without indices) denote query terms.*

– **Root Elimination:**

$$
\begin{array}{lll}
(1) & l \preceq l\{t_1^2, \ldots, t_m^2\} \Leftrightarrow \mathit{true} & \mathit{if}\ m \geq 1 \\
& l \preceq l\{\} \Leftrightarrow \mathit{true} & \\
& l\{\} \preceq l\{t_1^2, \ldots, t_m^2\} \Leftrightarrow \mathit{false} & \mathit{if}\ m \geq 1 \\
& l\{\} \preceq l \Leftrightarrow \mathit{true} & \\
& l\{\} \preceq l\{\} \Leftrightarrow \mathit{true} &
\end{array}
$$

$$
\begin{array}{lll}
(2) & l\{\{t_1^1, \ldots, t_n^1\}\} \preceq l \Leftrightarrow \mathit{false} & \mathit{if}\ n \geq 1 \\
& l\{\{t_1^1, \ldots, t_n^1\}\} \preceq l\{\} \Leftrightarrow \mathit{false} & \mathit{if}\ n \geq 1 \\
& & \\
& l\{t_1^1, \ldots, t_n^1\} \preceq l \Leftrightarrow \mathit{false} & \mathit{if}\ n \geq 1 \\
& l\{t_1^1, \ldots, t_n^1\} \preceq l\{\} \Leftrightarrow \mathit{false} & \mathit{if}\ n \geq 1
\end{array}
$$

$$
\begin{array}{ll}
(3)\ \textit{Let } \Pi \textit{ be the set of total functions } \{t_1^1, \ldots, t_n^1\} \to \{t_1^2, \ldots, t_m^2\}\textit{:} & \\
\quad l\{\{t_1^1, \ldots, t_n^1\}\} \preceq l\{t_1^2, \ldots, t_m^2\} \Leftrightarrow \bigvee_{\pi \in \Pi} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq \pi(t_i^1) & \\
\qquad \mathit{if}\ n, m \geq 1 &
\end{array}
$$

$$
\begin{array}{ll}
\textit{Let } \Pi \textit{ be the set of total, surjective functions } \{t_1^1, \ldots, t_n^1\} \to \{t_1^2, \ldots, t_m^2\}\textit{:} & \\
\quad l\{t_1^1, \ldots, t_n^1\} \preceq l\{t_1^2, \ldots, t_m^2\} \Leftrightarrow \bigvee_{\pi \in \Pi} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq \pi(t_i^1) & \\
\qquad \mathit{if}\ n, m \geq 1 &
\end{array}
$$

$$
\begin{array}{lll}
(4) & l_1\{\{t_1^1, \ldots, t_n^1\}\} \preceq l_2\{t_1^2, \ldots, t_m^2\} \Leftrightarrow \mathit{false} & \mathit{if}\ l_1 \not= l_2\ (n, m \geq 0) \\
& & \\
& l_1\{t_1^1, \ldots, t_n^1\} \preceq l_2\{t_1^2, \ldots, t_m^2\} \Leftrightarrow \mathit{false} & \mathit{if}\ l_1 \not= l_2\ (n, m \geq 0)
\end{array}
$$

– $\rightsquigarrow$ **Elimination:**

$$X \rightsquigarrow t^1 \preceq t^2 \quad \Leftrightarrow \quad t^1 \preceq t^2 \ \wedge\ t^1 \preceq X \ \wedge\ X \preceq t^2$$

– **Descendant Elimination:**

$$\mathit{desc}\ t^1 \preceq l_2\{t_1^2, \ldots, t_m^2\} \Leftrightarrow t^1 \preceq l_2\{t_1^2, \ldots, t_m^2\} \vee \bigvee_{1 \leq i \leq m} \mathit{desc}\ t^1 \preceq t_i^2 \quad \mathit{if}\ m \geq 0$$

Applying the $\rightsquigarrow$ and descendant elimination rules to a constraint store C in disjunctive normal form may yield a constraint store not in disjunctive normal form. Thus, the method has to restore from time to time the disjunctive normal form of C. In doing so, the formulas *true* and *false* are treated as usual: *true* is removed from conjunctions, conjunctions containing *false* are removed.

In the following, $mgcu(t_1, \ldots, t_n)$ (with $t_1, \ldots, t_n$ construct terms) returns a most general commutative-unifier of $t_1, \ldots, t_n$ (in the sense of [13]) expressed as either *false*, if t_1 and t_2 are not commutative-unifiable, or as *true* if t_1 and t_2 are commutative-unifiable and do not contain variables, or else as a conjunction of

equations of the form $X = t$. Note that most general commutative-unifiers are only computed for construct terms (i.e. terms without $\rightsquigarrow$ and *desc* construct). Recall that commutative unification is decidable.

In the definition below, simulation unification is initialised with $X_0 \rightsquigarrow t^q \preceq t^c$, where X_0 is a variable occurring neither in t^q nor in t^c, instead of simply $t^q \preceq t^c$. The additional variable X_0 serves to a complete specification of the answers returned. This is useful in proving the correctness of simulation unification but can usually be dispensed of in practice.

Definition 15 (Simulation Unification).

1. **Initialisation:** $C := X_0 \rightsquigarrow t^q \preceq t^c$
   (with t^q query term, t^c construct term and t^q, t^c and X_0 variable disjoint).
2. **Term Decomposition:**
   Until C can no longer be modified, repeat performing one of:
   - *Apply a (applicable) Term Decomposition rule to C*
   - *Put C in disjunctive normal form*
3. **Variable Binding:**
   Replace each $X \preceq t$ in C with $X = t$.
4. **Consistency Verification:**
   For each disjunct D of C and for each variable X occurring in D do:
   Replace in D the equations $X = t_1, \ldots, X = t_n$ by $mgcu(t_1, \ldots, t_n)$.

For efficiency reasons it is preferable to intertwine the Term Decomposition and Consistency Verification phases instead of performing them one after another. The sequential processing in Definition 15 simplifies the proofs.

Proposition 1 (Correctness and Completeness). *Let t^q be a query term, t^c a construct term, and X_0 a variable such that t^q, t^c and X_0 are variable disjoint. There exists a substitution τ such that $t^q\tau$ and $t^c\tau$ are database terms and $t^q\tau = t^c\tau$ if and only if a simulation unification initialised with $X_0 \rightsquigarrow t^q \preceq t^c$ returns a substitution σ such that*

- *For each variable X in t^q, $X\sigma$ is a subterm of $t^q\sigma$.*
- *$t^q\tau$ is an instance of $t^q\sigma$.*
- *$t^c\tau$ is an instance of $t^c\sigma$.*

The proof of Proposition 1 is given in the full version of this paper [8].

3.5 Examples

$f\{\{X \rightsquigarrow b, Y \rightsquigarrow b\{\{c,d\}\}\}\}$ and $f\{a, b\{c,d,e\}, b\{e\}\}$ "simulation unify" yielding the following constraints: $(X = b\{c,d,e\} \wedge Y = b\{c,d,e\}) \vee (X = b\{e\} \wedge Y = b\{c,d,e\})$.

Also, the terms $X \rightsquigarrow desc\ (Y \rightsquigarrow f\{\{a\}\})$ and $g\{f\{Z,b,c\}, h\{f\{a,b\}\}\}$ "simulation unify" yielding $((Y = f\{Y,b,c\} \wedge a \preceq Z) \vee Y = f\{a,b\}) \wedge X = g\{f\{Z,b,c\}, h\{f\{a,b\}\}\}$.

The steps of these simulation unifications are given in the full version [8] of this paper. Note that these simulation unifications constrain variables "on both sides", i.e. simulation unification is no matching but a full-fledged unification.

4 Related Work and Conclusion

The articles [14,15,16] have already pointed out the drawbacks of relying on a navigational node selection à la XPath [17] and XQuery [2] for query and transformation languages for XML and semistructured data.

The language UnQL [11] has introduced simulation as a means for query answering. UnQL, like Xcerpt, uses the notions of patterns and templates. UnQL and Xcerpt differ from each other as follows. First, a query in UnQL consists of a single "select-where" expression which can be processed with pattern matching. In contrast, a query in Xcerpt might "chain" several "construct-query rules" requiring a "unification" which is capable of binding variables from both of the terms to be "unified". Second, variables in UnQL can only occur as leaves of query patterns. Complex queries might require the use of several patterns in UnQL, where a single pattern suffices in Xcerpt.

In [14] a language for querying and transforming semistructured data is described. Like XPath and XQuery this language has variables for nodes, i.e. in the Xcerpt terminology labels.

[15] describes fxt, a language for querying and transforming semistructured data. fxt has variables for terms (or trees) and forests. fxt offers regular expressions similar to those of XPath for node selection. In contrast, the approach proposed in the present paper uses like Prolog variables for subterms. Arguably, languages with term variables makes data description less navigational than languages with node variables.

The language semantics in [14] is based upon a so-called component calculus and an algebra, very much in the style of XQuery's algebra which is inspired from functional languages. The language semantics given in [15] for fxt is in terms of tree automata. Arguably, Definition 13 is closer to a Tarski's style model theory and might therefore be seen as a more declarative semantics.

Several articles propose inference methods either rule-based or based upon consistency verification for XML data. [16] proposes a rule language very similar to Prolog called nowadays RuleML [18]. Several approaches that are too numerous for being explicitly mentioned here adapt techniques from feature logics to XML data. These approaches are usually named referring to "ontology" and/or "Semantic Web". Common to RuleML and the ontology or Semantic Web approaches is that the language they propose do not support a direct access to XML data. Instead, their languages require a translation into a specific syntax. In some cases, like the binary predicate language RDF, this syntax might seem too stringent. For the authors of this paper, a direct access to XML data is an essential feature of an inference language for Web-based databases and semantic reasoning with Web data.

Simulation is no new notion. It is commonly used in process algebra and graph theory. It has been applied to semistructured data e.g. in [10,19,3] for schema validation. Graph simulation in general has been studied extensively cf. [9,20] (simulation is called "path inclusion" in [20]).

Several unification methods have been proposed that, like simulation unification, process flexible terms or structures, notably feature unification [21,22]

and associative-commutative-unification, short AC-unification, [23]. Simulation unification differs from feature unification in several aspects (discussed in [8]).

Simulation unification might remind of theory unification [24]. The significant difference between both is that simulation unification is based upon an order relation, while theory unification refers to a congruence relation.

There are interesting similarities between simulation unification and approaches to constraint solving over finite domains [25]. Simulation unification relies on a possibly disjunctive constraint store. This is rarely the case for constraint solvers. However, constraint programming approaches such as aggregation constraints [26] and constructive disjunction [27] seem interesting techniques for the future development of the language Xcerpt.

In this paper, a novel approach to querying and transforming XML and semistructured data based has been outlined. This approach is based on logic programming and a novel form of unification, simulation unification. A few aspects of a language under development, Xcerpt, have been presented. Many issues deserve further investigations. In particular, the complexity of simulation unification and its efficient implementation deserve further research.

Acknowledgements

The authors are thankful to Slim Abdennadher and Norbert Eisinger for useful suggestions.

References

1. W3C http://www.w3.org/Style/XSL/: Extensible Stylesheet Language (XSL). (2000) 255
2. W3C http://www.w3.org/TR/xquery/: XQuery: A Query Language for XML. (2001) 255, 256, 268
3. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web. From Relations to Semistructured Data and XML . Morgan Kaufmann Publishers, San Francisco, CA (2000) 256, 258, 262, 268
4. Fernandez, M., Siméon, J., Wadler, P.: XML Query Languages: Experiences and Examplars. Communication to the XML Query W3C Working Group (1999) 256
5. Alashqur, A. M., Su, S. Y. W., Lam., H.: OQL: A Query Language for Manipulating Object-Oriented Databases. In: Proc. 15th Int. Conf. on Very Large Data Bases (VLDB). (1989) 256
6. Chamberlin, D., Fankhauser, P., Marchiori, M., Robie, J.: XML Query Use Cases. W3C Working Draft 20 (2001) 256
7. Bry, F., Schaffert, S.: Pattern Queries for XML and Semistructured Data. Technical Report PMS-FB-2002-5, Inst. for Computer Sciences, University of Munich, http://www.pms.informatik.uni-muenchen.de/publikationen/#PMS-FB-2002-5 (2002) 260
8. Bry, F., Schaffert, S.: Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. Technical Report PMS-FB-2002-2, http://www.pms.informatik.uni-muenchen.de/publikationen/#PMS-FB-2002-2 (2002) 261, 262, 267, 269

9. Henzinger, M. R., Henzinger, T. A., Kopke, P. W.: Computing Simulations on Finite and Infinite Graphs (1996) 262, 268
10. Fernandez, M., Suciu, D.: Optimizing Regular Path Expressions Using Graph Schemas. In: Proceedings of the Int. Conf. on Data Engineering. (1988) 14–23 262, 268
11. Buneman, P., Fernandez, M., Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. VLDB Journal **9** (2000) 76–110 262, 268
12. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37** (1998) 95–138 265
13. Baader, F.: Unification in Commutative Theories. In: Unification. Academic Press (1989) 417–435 266
14. Grahne, G., Lakshmanan, L. V. S.: On the Difference between Navigating Semistructured Data and Querying It. In: Workshop on Database Programming Languages. (1999) 268
15. Berlea, A., Seidl, H.: fxt – A Transformation Language for XML Documents. Journal of CIT, Special Issue on Domain-Specific Languages (2001) 268
16. Boley, H.: Relationships Between Logic Programming and XML. In: Proc. 14th Workshop Logische Programmierung, Würzburg (2000) 268
17. W3 Consortium http://www.w3.org/TR/xpath: XML Path Language (XPath). (1999) 268
18. DFKI: RuleML – Rule Markup Language. http://www.dfki.uni-kl.de/ruleml/ (2002) 268
19. Buneman, P., Davidson, S. B., Fernandez, M. F., Suciu, D.: Adding Structure to Unstructured Data. In: Proceedings of ICDT'97. Volume 1186., Springer (1997) 336–350 268
20. Kilpeläinen, P.: Tree Matching Problems with Applications to Structured Text Databases. PhD thesis, Dept. of Computer Sciences, University of Helsinki (1992) 268
21. Aït-Kaci, H., Podelski, A., Goldstein, S. C.: Order-Sorted Theory Unification. Technical Report 32, digital – Paris Research Laboratory (1993) 268
22. Smolka, G.: Feature Constraint Logics for Unification Grammars. Journal of Logic Programming **12** (1992) 51–87 268
23. Fages, F.: Associative-Commutative Unification. In: Proc. 7th Int. Conf. on Automated Deduction (Napa, CA). Volume 170., Berlin, Springer (1984) 194–208 269
24. Baader, F., Snyder, W.: Unification Theory. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier Science Publishers (1999) 269
25. Montanari, U., Rossi, F.: Finite domain constraint solving and constraint logic programming. In Benhamou, F., Colmerauer, A., eds.: Constraint Logic Programming: Selected Research. MIT press (1993) 201–221 269
26. Ross, K. A., Srivastava, D., Stuckey, P. J., Sudarshan, S.: Foundations of aggregation constraints. Theoretical Computer Science B **190** (1994) 269
27. Würtz, J., Müller, T.: Constructive disjunction revisited. In: KI - Künstliche Intelligenz. (1996) 377–386 269

A Proof-Theoretic Foundation for Tabled Higher-Order Logic Programming

Brigitte Pientka*

Department of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, USA
bp@cs.cmu.edu

Abstract. Higher-order logic programming languages such as *Elf* extend first-order logic programming in two ways: first-order terms are replaced with (dependently) typed λ-terms and the body of clauses may contain implication and universal quantification. In this paper, we describe *tabled higher-order logic programming* where some redundant computation is eliminated by memoizing sub-computation and re-using its result later. This work extends Tamaki and Sato's search strategy based on memoization to the higher-order setting. We give a proof-theoretic characterization of tabling based on uniform proofs and prove soundness of the resulting interpreter. Based on it, we have implemented a prototype of a tabled logic programming interpreter for *Elf*.

1 Introduction

Tabled first-order logic programming has been successfully applied to solve complex problems such as implementing recognizers and parsers for grammars [25], representing transition systems CCS and writing model checkers [6]. The idea behind it is to eliminate redundant computation by memoizing sub-computation and re-using its results later. The resulting search procedure is complete and terminates for programs with the bounded-term size property. The XSB system [22], a tabled logic programming system, demonstrates impressively that tabled together with non-tabled programs can be executed efficiently.

Higher-order logic programming languages such as *Elf* [14] extend first-order logic programming in two ways: first-order terms are replaced with dependently typed λ-terms and the body of clauses may contain implication and universal quantification. It offers a generic framework for 1) implementing logical systems as *Elf* programs, 2) executing them and generating a certificate for each execution via an interpreter 3) checking certificates via type-checking and 4) reasoning with and about logical systems via a meta-level theorem prover *Twelf* [19]. One of its applications lies in "certifying code" where programs are equipped with a certificate (proof) that asserts certain safety properties. The safety policy can be represented as a higher-order logic program in *Elf*. Appel and Felty [1] use the

* This work was partially supported by NSF Grant CCR-9988281.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 271–286, 2002.
© Springer-Verlag Berlin Heidelberg 2002

logic programming interpreter to execute the specification and generate a certificate that a given program fulfills a specified safety policy. Necula and Rahul [12] use a logic programming interpreter for checking the correctness of a certificate. In their case, the certificate is a bit-string that guides the logic programming interpreter to resolve non-deterministic choices. Representing and executing different safety policies using *Elf* reduces the effort required for each specific policies and offers an ideal environment for experimenting and combining safety policies. Proof search based on logic programs plays a central role in this setting, but redundant computation may hamper the performance and computation may not terminate, although the underlying domain is finite.

In this paper, we present *tabled higher-order logic programming* where some redundant computation is eliminated by memoizing sub-computation and reusing its result later. As higher-order logic programming allows nested implications and universal quantification in the body of clauses, goals might depend on a context of assumptions. We also have dependencies among terms, as the term language is derived from the dependently typed λ-calculus. The combination of both requires careful design of the table and table operations. We give a proof-theoretic characterization of tabled higher-order logic programming based on uniform proofs [10] and show soundness of the resulting interpreter. This work forms the basis of the implemented tabled interpreter for the language *Elf*. Although we concentrate on the logical framework LF, which is the basis of *Elf*, it seems possible to apply the presented approach to λProlog [11] or Isabelle [13], which are based on hereditary Harrop formulas and simply typed terms.

The paper is organized as follows: In Sec. 2 we introduce a type system for Mini-ML including subtyping. Using this example, we review briefly tabled logic programming and discuss higher-order tabled computation in Sec. 3. In Sec. 4 we review uniform proofs and then develop a tabled uniform proof system and prove soundness. In Sec. 5 we discuss related work and summarize the results.

2 A Motivating Example: Subtyping

2.1 Background

As a running example we consider a type system for a restricted functional language Mini-ML, which includes subtyping. We only consider a small set of expressions, negative numbers $\mathsf{n}(e)$, natural numbers z and $\mathsf{s}(e)$, functions $\mathsf{lam}\ x.e$, function application $\mathsf{app}\ e_1\ e_2$. The type zero contains only the number z, the type pos represents all positive natural number and the type nat describes all natural numbers; the type neg denotes the negative numbers and the type int describes all numbers.

$$\begin{array}{rl} e & ::= \mathsf{n}(e) \mid \mathsf{z} \mid \mathsf{s}(e) \mid \mathsf{lam}\ x.e \mid \mathsf{app}\ e_1\ e_2 \\ \tau & ::= \mathsf{neg} \mid \mathsf{zero} \mid \mathsf{pos} \mid \mathsf{nat} \mid \mathsf{int} \mid \tau_1 \rightarrow \tau_2 \end{array}$$

The specification of the subtyping relation using reflexivity and transitivity and the typing rules are straightforward (see Fig. 1). For a full description we refer the reader to [20].

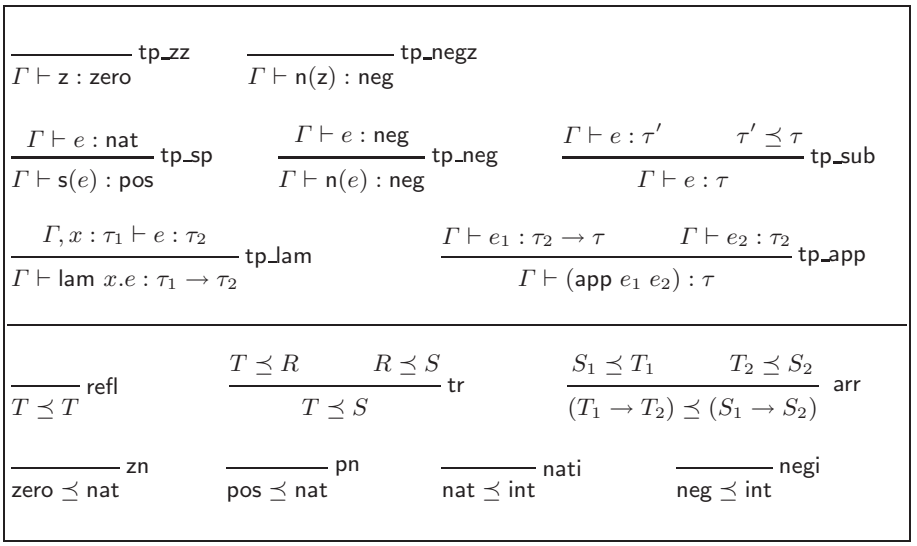


Fig. 1. Typing rules including subtyping relation

The subtyping relation is directly translated into *Elf* using logic programming notation. Constants `neg`, `zero`, `pos`, `nat` and `int` represent the basic types and the function type is denoted by `T1 => T2`. Throughout this example, we reverse the arrow $A_1 \to A_2$ writing instead $A_2 \leftarrow A_1$. From a logic programming point of view, it might be more intuitive to think of the clause $H \leftarrow A_1 \leftarrow \ldots \leftarrow A_n$ as $H \leftarrow A_1\ ,\ \ldots\ ,\ A_n$.

```
refl  :sub T T.      zn    :sub zero nat. arr:sub (T1 => T2) (S1 => S2)
tr    :sub T S       pn    :sub pos nat.      <- sub S1 T1
       <- sub T R    negi  :sub neg int.      <- sub T2 S2
       <- sub R S.   nati  :sub nat int.

tp_sub:of E T        tp_lam:of (lam ([x] E x)) (T1 => T2)
       <- of E T'           <- ({x:exp} of x T1 -> of (E x) T2).
       <- sub T' T.
```

For implementing the subtyping relations logic programming based on Horn clauses suffices. However, *Elf* is much richer than first-order logic programming and also supports elegant encodings based on higher-order abstract syntax [18]. Variables bound in constructors such as lam will be bound with λ in *Elf*. The binding described by λ-expression $\lambda x.Ex$ is denoted by `[x] E x` using *Elf* syntax and the Mini-ML expression lam $x.e$ is represented as `lam [x] E x` in *Elf*. Substitution is modeled via application and β-reduction. In addition to the variable binding construct, *Elf* supports reasoning from hypotheses and handling parameters. The premise of typing rule for lam depends on the new parameter x and the hypothesis that x is of type τ_1. Moreover, we assume that it is possible to re-

name all variables in e, if necessary. In *Elf* this is represented by `({x:exp} of x T1 -> of (E x) T2)` where `{x:exp}` denotes the universal quantifier Πx:exp. We can show `of (lam ([x] E x)) (T1 => T2)`, if we can prove that for a new variable `x`, if `x` has type `T1` then the body of the function `(E x)` has type `T2`. For a more detailed discussion see [16]

Higher-order logic programming suffers from the same problems as first-order logic programming: computation may be trapped in infinite paths and performance may be hampered by redundant computation. For example, the execution of the query `sub zero T` will end in an infinite branch trying to apply the transitivity rule. Similarly, the execution of the query `of (lam [x] x) T` will not terminate and fail to enumerate all possible types. In addition, we repeatedly type-check sub-expressions, which occur more than once. To eliminate redundancy, some sophisticated type checkers for example for refinement types memoize the result of sub-computations to obtain more efficient implementations. In this paper, we extend higher-order logic programming languages such as *Elf* with generic memoization techniques, called *tabled higher-order logic programming*. This has several advantages. Although it is possible to derive an algorithmic subtyping relation for the given example, this might not be trivial in general. To refine the implementation further by adding explicit support for memoization, complicates the type checker. As a consequence, the certificates, which are produced as a result of the execution, are larger and contain references to the explicit memoization data-structure. This is especially undesirable in the context of certified code where certificates are transmitted to and checked by a consumer, as sending larger certificates takes up more bandwidth and checking them takes more time. Moreover, proving the correctness of the type-checker with special memoization support will be hard, because we need to reason explicitly about the structure of memoization. As tabled logic programming terminates for programs with the bounded term-size property, we are also able to disprove certain statements. This in turn helps the user to debug the specification and implementations. In this paper, we propose to extend higher-order logic programming with memoization techniques.

2.2 Tabled Logic Programming: Review

Tabling methods evaluate programs by maintaining tables of subgoals and their answers and by resolving repeated occurrences of subgoals against answers from the table. We review briefly Tamaki and Sato's multi-stage strategy [23], which differs only insignificantly from SLG resolution [5] for programs without negation. To demonstrate tabled computation, we consider the evaluation of the query sub zero T in more detail.

The search proceeds in multiple stages. The table serves two purposes: 1) We record all sub-goals encountered during search. If the current goal is not in the table, then we add it to the table and proceed with the computation. Computation at a node is suspended, if the current goal is a variant of a table entry. 2) In addition to the sub-goals we are trying to solve, we also store the results of computation in the table as a list of answers to the sub-goals. To simplify the

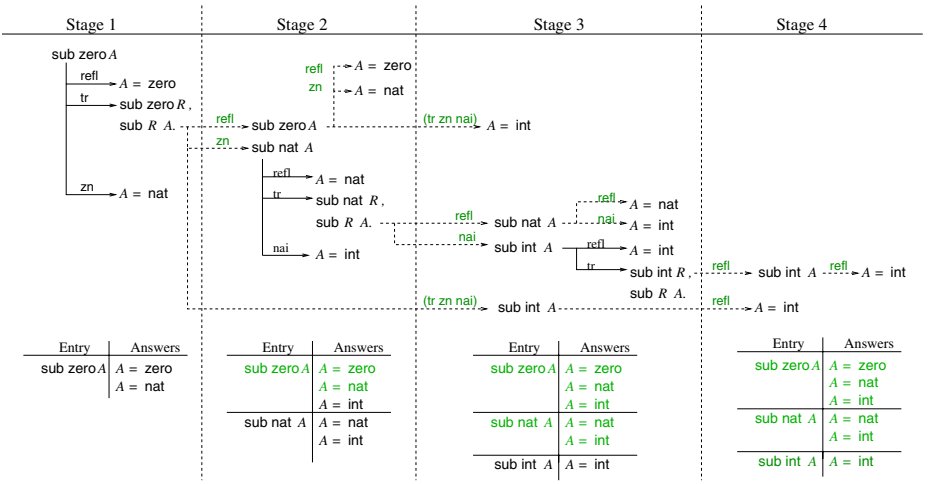


Fig. 2. Staged computation

table in this presentation, we do not record the certificate (proof term) explicitly in the table, although we do record it in the actual implementation. In each stage we apply program clauses and answers from the table. Figure 2 illustrates the search process.

The root of the search tree is labeled with the goal sub zero A. Each node is labeled with a goal statement and each child node is the result of applying a program clause or an answer from the table to the *leftmost* atom of the parent node. Applying a clause $H \leftarrow A_1 \leftarrow A_2 \ldots \leftarrow A_n$ results in the subgoals $A_1, A_2, \ldots, A_n$ where all of these subgoals need to be satisfied. We will then expand the first subgoal A_1 carrying the rest of the subgoals $A_2, \ldots, A_n$ along. If a branch is successfully solved, we show the obtained answer. To distinguish between program clause resolution and re-using of answers, we have two different kinds of edges in the tree. The edges obtained by program clause resolution are solid while edges obtained by reusing answers from the table are dashed. Both are labeled with the clause name that was used to derive the child node. Using the labels at the edges we can reconstruct the proof term for a given query. In general, we will omit the actual substitution under that the parent node unifies with the program clause to avoid cluttering the example. To ensure we generate all possible answers for the query, we restrict the re-use of answers from the table. In each stage, we are only allowed to re-use answers that were generated in previous stages. Answers from previous stages (available for answer resolution) are marked gray, while current answers (not available yet) are black.

3 Tabled Higher-Order Logic Programming

In tabled higher-order logic programming, we extend tabling to handle subgoals that may contain implications and universal quantification and our term

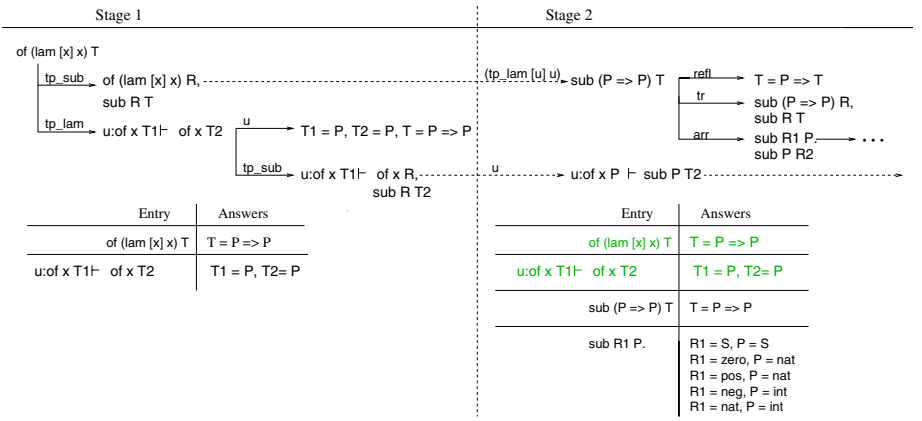


Fig. 3. Staged computation for identity function

language is the dependently typed λ-calculus. The table entries are no longer atomic goals, but atomic goals A together with a context Γ of assumptions. In addition, terms might depend on assumptions on Γ. To highlight some of the challenges we present the evaluation of the query $\mathsf{of}\ (\mathsf{lam}\ [x]\ x)\ T$ in Fig. 3.

The possibility of nested implications and universal quantifiers adds a new degree of complexity to memoization-based computation. Retrieval operations on the table need to be redesigned. One central question is how to look up whether a goal $\Gamma \vdash a$ is already in the table. There are two options: In the first option we only retrieve answers for a goal a given a context Γ, if the goal together with the context matches an entry $\Gamma' \vdash a'$ in the table. In the second option we match the subgoal a against the goal a' of the table entry $\Gamma' \vdash a'$, and treat the assumptions in Γ' as additional subgoals, thereby delaying satisfying these assumptions. We choose the first option of retrieving goals together with their dynamic context Γ'. One reason is that it restricts the number of possible retrievals early on in the search. For example, to solve subgoal $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ R, \mathsf{sub}\ R\ T_2$, we concentrate on solving the left-most goal $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ R$ keeping in mind that we still need to solve $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{sub}\ R\ T_2$. As there exists a table entry $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ T_2$, which is a variant of the current goal $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ R$, computation is suspended.

Due to the higher-order setting, the predicates and terms might depend on Γ. Virga [24] developed in his PhD thesis techniques, called subordination, to analyze dependencies in *Elf* programs statically before execution. In the Mini-ML example, the terms of type `exp` and `tp` are independent of each other. On the level of predicates, the type checker `of` depends on the subtyping relation `sub`, but not vice versa. When checking whether a subgoal $\Gamma \vdash a$ is already in the table, we exploit the subordination information in two ways. First, we use it to analyze the context Γ and determine which assumptions might contribute to the proof of a. For example the proof for $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ T_2$ depends on the assumption u. However, the proof for $u{:}\mathsf{of}\ x\ P \vdash \mathsf{sub}\ P\ T_2$ cannot depend on the

assumption u, as the predicate sub does not refer to the predicate of . Therefore, when checking whether $u{:}\mathsf{of}\ x\ P \vdash \mathsf{sub}\ P\ T_2$ is already in the table, it suffices to look for a variant of $\mathsf{sub}\ P\ T_2$. In the given example, computation at subgoal $u{:}\mathsf{of}\ x\ P \vdash \mathsf{sub}\ P\ T_2$ is suspended during stage 2 as the table already contains $\mathsf{sub}\ R_1\ P$. If we for example first discover $u{:}\mathsf{of}\ x\ P \vdash \mathsf{sub}\ P\ T_2$, then we store the strengthened goal $\mathsf{sub}\ P\ T_2$ in the table with an empty context.

Second, subordination provides information about terms. As we are working in a higher-order setting, solutions to new existential variables, which are introduced during execution, might depend on assumptions from Γ. For example, applying the subtyping rule to $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ T_2$ yields the new goal $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ (R\ x\ u)\ ,\ \mathsf{sub}\ (R\ x\ u)\ T_2$ where the solution for the new variable R might depend on the new variable $x{:}\,\mathsf{exp}$ and the assumptions $u{:}\mathsf{of}\ x\ T_1$. However, we know that the solution must be an object of $\mathtt{tp}$ and that objects of $\mathtt{tp}$ are independent of Mini-ML expressions $\mathtt{exp}$ and the Mini-ML typing rules $\mathtt{of}$. Hence, we can omit x and u and write $u{:}\mathsf{of}\ x\ T_1 \vdash \mathsf{of}\ x\ R,\ \mathsf{sub}\ R\ T_2$. Before comparing goals with table entries and adding new table entries, we eliminate unnecessary dependencies from the subgoal $\Gamma \vdash a$. This allows us to detect more loops in the search tree and eliminate more redundant computation. For further discussion issues in higher-order tabling, we refer the interested reader to [20].

4 A Foundation for Tabled Higher-Order Logic Programming

4.1 Uniform Proofs

Computation in logic programming is achieved through proof search. Given a goal (or query) A and a program Γ, we derive A by successive application of clauses of the program Γ. Miller *et al* [10] propose to interpret the connectives in a goal A as *search instructions* and the clauses in Γ as specifications of how to continue the search when the goal is atomic. A proof is *goal-oriented* if every compound goal is immediately decomposed and the program is accessed only after the goal has been reduced to an atomic formula. A proof is *focused* if every time a program formula is considered, it is processed up to the atoms it defines without need to access any other program formula. A proof having both these properties is *uniform* and a formalism such that every provable goal has a uniform proof is called an *abstract logic programming language*.

Elf is one example of an abstract logic programming language, which is based on the LF type theory. Π-quantifier and $\rightarrow$ suffice to describe LF. In this setting types are interpreted as clauses and goals and typing context represents the store of program clauses available. We will use types and formulas interchangeably. Types and programs are defined as follows;

$$\begin{array}{lll} \text{Types } A & ::= a \mid A_1 \rightarrow A_2 \mid \Pi x : A_1.A_2 & \text{Terms } M ::= H \cdot S \mid \lambda x : A.M \\ \text{Programs } \Gamma & ::= \cdot \mid \Gamma, x : A & \text{Spines } S ::= \textsc{nil} \mid M; S \\ & & \text{Heads } H ::= c \mid x \end{array}$$

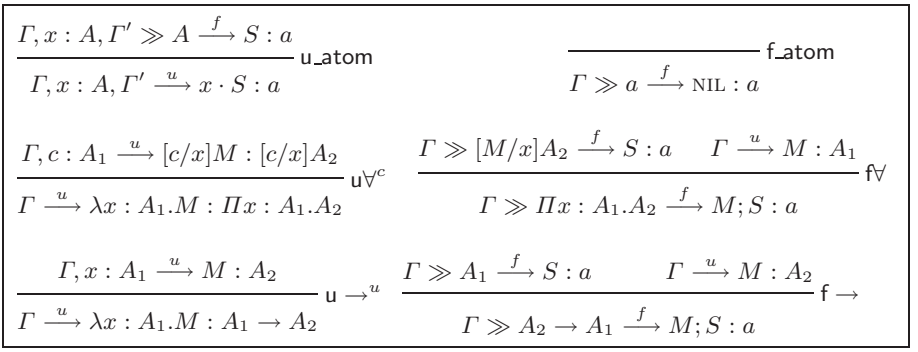


Fig. 4. Uniform deduction system for $\mathcal{L}$

a ranges over atomic formulas. The function type $A_1 \to A_2$ corresponds to an implication. The Π-quantifier, denoting dependent function type, can be interpreted as the universal $\forall$-quantifier. The clause `tr:sub T S <- sub T R <- sub R S.` is interpreted as $\mathsf{tr}{:}\Pi t{:}\mathsf{tp}.\Pi s{:}\mathsf{tp}.\Pi r{:}\mathsf{tp}.\ \mathsf{sub}\ r\ s \to (\mathsf{sub}\ t\ r \to \mathsf{sub}\ t\ s)$. Every type has a corresponding proof term M and we assume that all proof terms are in normal form. In the example from Sec. 2, the proof term corresponding to sub zero int is given as tr zn refl. Note that we actually omitted the implicit arguments zero and nat, which denote the instantiation of transitivity rule. In the following discussion, we will include implicit arguments in the proof term representation. To represent proof terms, we use the spine notation [4]. To give an intuition for this notation we give a few examples. tr zero nat nat zn refl is denoted using spine notation by $\mathsf{tr} \cdot \mathsf{zero}\ ;\ \mathsf{nat}\ ;\ \mathsf{nat}\ ;\ \mathsf{zn}\ ;\ \mathsf{refl}\ ;\ \text{NIL}$. The proof term $\mathsf{tp_lam}\ T\ (\lambda x{:}\exp.x)\ T\ (\lambda x{:}\exp.\lambda u{:}\mathsf{of}\ x\ T.\ u)$ is denoted by $\mathsf{tp_lam} \cdot T\ ;\ (\lambda x{:}\exp.x \cdot \text{NIL})\ ;\ T\ ;\ (\lambda x{:}\exp.\lambda u{:}\mathsf{of}\ x\ T.\ u \cdot \text{NIL})\ ;\ \text{NIL}$.

We can characterize uniform proofs by two main judgments: $\Gamma \xrightarrow{u} M : A$ says there is a uniform proof M for A from the program Γ and $\Gamma \gg A \xrightarrow{f} S : a$ there exists a focused proof S for the atom a by focusing on program clause A. Taking a type-theoretic view, we can interpret the first judgment as M has type A in the context Γ and the later as S has type a in context Γ. Inference rules describing uniform and focused proofs are given in Fig. 4. In the rule f$\forall$, we instantiate the bound variable x with a term M. As x has type A_1, we check that M has type A_1 in Γ. Miller [9] shows for the simply-typed λ-calculus that if M is a solution for x in the context Γ then there exists a solution M' of type $\Pi\Gamma.A_1$ such that $M' \cdot \Gamma$ is also a solution for x and M' is well-typed in the empty context. We write $\Pi\Gamma.A_1$ for the type $\Pi x_1{:}B_1. \ldots \Pi x_n{:}B_n.A_1$ where Γ is a context $x_1{:}B_1, \ldots, x_n{:}B_n$ and $M' \cdot \Gamma$ as an abbreviation for $M' \cdot x_1; \ldots; x_n; \text{NIL}$. Moreover, there is a one-to-one correspondence between these two solutions, as $M' \cdot \Gamma$ reduces to M. Following Miller's terminology, we say M' is the result of raising M. Pfenning [15] investigated this notion in the setting of the calculus of construction, which includes LF. Intuitively, M depends globally on the assumptions in Γ. Raising allows us to localize dependencies by replacing M with $M' \cdot \Gamma$.

This step becomes important when we want to compute the instantiation of x in the f$\forall$ rule by introducing a existential variable X and instantiating X by unification. We come back to it in the next section. To reflect raising, we rewrite the f$\forall$ rule to:

$$\frac{\Gamma \gg [M' \cdot \Gamma/x]A_2 \xrightarrow{f} S : a \qquad \cdot \xrightarrow{u} M' : \Pi\Gamma.A_1}{\Gamma \gg \Pi x : A_1.A_2 \xrightarrow{f} (M' \cdot \Gamma); S : a}\ \text{f}\forall$$

The proof term represents the witness of the proof. When searching for a uniform proof, the proof term is constructed simultaneously. In the following discussion, we will not mention proof terms explicitly, but keep in mind that they are silently generated as a result of the proof.

4.2 Uniform Proofs with Answer Substitutions

The result of a computation in logic programming is generally an answer substitution θ for the existentially quantified variables in a goal A. To obtain an algorithm that computes answer substitutions, we substitute existential variables X for the bound variable x in the f$\forall$ rule. In fact, we replace M' with X in the raised version of f$\forall$ rule. Note that X does not globally depend on Γ anymore as raising allowed us to rotate the existentially quantified variables to the outside. As all dependencies are local, we can model dependencies between parameters and existential variables by annotating existential variables X with their type. Existential variables are instantiated later during unification yielding a substitution θ. An alternative would be to use mixed-prefixes [9] to model dependencies. However this would complicate the presentation further. We view the answer substitution θ as a collection of constraints to the existential variables in a goal A. In general, unification for higher-order terms is undecidable, however Pfenning showed that unification of higher-order patterns in the context of LF type theory is decidable and unitary[15]. Many programs fall into the decidable fragment and we concentrate on this case.

$$\begin{array}{ll} \text{Substitution } \theta ::= & \cdot \mid \theta, X_A = M \\ \text{Composition} & \cdot \circ \theta = \theta \\ & (\theta_1, X_A = M) \circ \theta_2 = (\theta_1 \circ \theta_2), X_{A[\theta_2]} = M[\theta_2] \end{array}$$

We require that all free (existential) variables X defined by a substitution are distinct. We write $\mathsf{dom}(\theta)$ for the free variables defined by a substitution and $\mathsf{codom}(\theta)$ for all the free variables occurring in the term M. For a *ground substitution* $\mathsf{codom}(\theta)$ is empty. We write $M[\theta]$, $A[\theta]$, and $\Gamma[\theta]$ for the application of a substitution to a term, proposition or context. *Composition*, written as $\theta_1 \circ \theta_2$, has the property that $M[\theta_1 \circ \theta_2] = (M[\theta_1])[\theta_2]$ and similarly for propositions and contexts. In order for composition of substitutions to be well-defined and have the desired properties we require that $\mathsf{dom}(\theta_1)$ and $\mathsf{dom}(\theta_2)$ are disjoint, but of course variables in the co-domain of θ_1 can be defined by θ_2. Moreover, we require

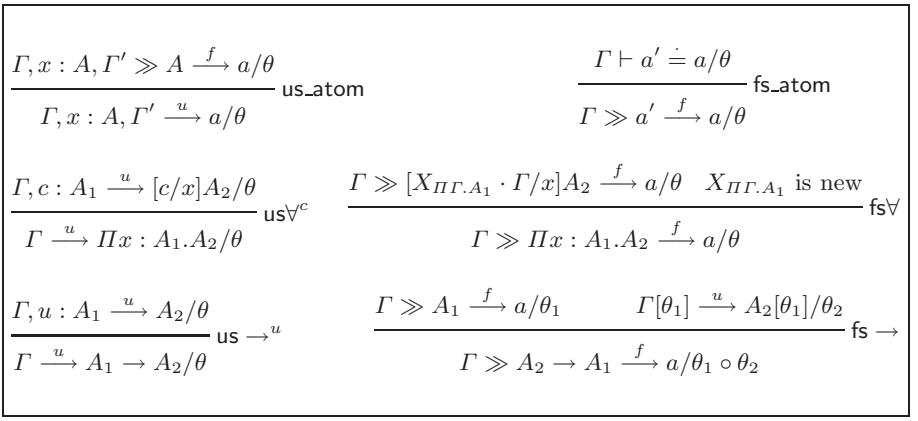


Fig. 5. Uniform deduction system for $\mathcal{L}_\theta$ with substitutions

that $\theta \circ \theta = \theta$. As an existential variable is annotated with its type A and A might itself contain existential variables, we need to apply the substitution θ_2 to M and to the type A during composition of substitutions. The two main judgments for computing answer substitutions are $\Gamma \xrightarrow{u} A/\theta$ and $\Gamma \gg A \xrightarrow{f} a/\theta$.

The inference rules are given in Fig. 5. To obtain an algorithm, we impose left-to-right order on the solution of the fs $\to$ rule. This matches our intuitive understanding of computation in logic programming. In the fs $\to$ rule for example we first decompose the focused clause until we reach the head of the clause. After we unified the head of the clause with our goal A on the right-hand side of the sequent and completed this branch, we proceed proving the subgoals. This left-to-right evaluation strategy only fixes a don't care non-deterministic choice in the inference system. In the fs$\forall$ rule we delay the instantiation of x by introducing a new existential variable X. In the fs_atom rule the instantiation for existentially quantified variables is obtained by unifying a with a' in the context Γ. θ is a solution to the unification problem $\Gamma \vdash a \doteq a'$ where a and a' are higher-order patterns.

There is still some non-determinism left in $\mathcal{L}_\theta$, which needs to be resolved in an actual implementation. In the us_atom rule, we do not specify which clause from Γ we pick and focus on. Logic programming interpreter usually try the clauses in the order they occur as backtracking over different choices is considered too expensive. This choice renders the search strategy incomplete in practice. However, the presented deductive system $\mathcal{L}_\theta$, which generates answer substitutions, is sound and complete, as expressed by the following two theorems.

Theorem 1 (Soundness). *If $\mathcal{D} : \Gamma \xrightarrow{u} A/\theta$ then for any grounding substitution σ' which grounds $\Gamma \xrightarrow{u} A/\theta$ we have $\mathcal{E}{:}\Gamma[\theta \circ \sigma'] \xrightarrow{u} A[\theta \circ \sigma']$.*

Theorem 2 (Completeness). *If $\mathcal{E} : \Gamma \xrightarrow{u} A[\sigma]$ for a grounding substitution σ then $\mathcal{D} : \Gamma \xrightarrow{u} A/\theta$ for some θ and $\sigma = \theta \circ \sigma'$ for some σ'.*

The proof requires a more general statement that also includes focused proofs, but otherwise is straightforward. In the next section, we extend this system $\mathcal{L}_\theta$ to include memoization.

4.3 Tabled Uniform Proofs

The idea behind tabled uniform proofs is to extend our two basic judgments with a table $\mathcal{T}$ in which we record atomic sub-goals and the corresponding answer substitutions and proof terms. A subgoal is a sequent $\Gamma \xrightarrow{u} a$ where Γ is a program context and a is an atomic goal, which we need to derive from Γ. When we discover the sub-goal $\Gamma \xrightarrow{u} a$ for the first time, we memoize this goal in the table. Note that the sequent $\Gamma \xrightarrow{u} a$ might potentially contain existential variables. Once we have proven the sub-goal $\Gamma \xrightarrow{u} a$, we add the answer substitution θ to the table. We keep in mind that we are silently generating proof terms together with answer substitution. We assume that some predicates are designated as tabled predicates where we record subgoals and corresponding answers. For predicates not designated as tabled the $\mathsf{us_atom}$ rule still applies.

Definition 1 (Table). *A table $\mathcal{T}$ is a collection of table entries. A table entry consists of two parts: a goal $\Gamma \xrightarrow{u} a$ and a list $\mathcal{A}$ of pairs, answer substitutions θ and proof terms M, such that $\Gamma[\theta] \xrightarrow{u} M[\theta] : a[\theta]$ is a solution.*

The table is a store of proven and still open conjectures. The open conjectures are the table entries that have an empty list of answers. The proven conjectures (lemmas) are the table entries that have a list of answer substitutions associated with them. As proof terms are generated and stored together with answer substitutions, we also have the actual proof for the given conjecture. We will design the inference rules in such a way that for any solution in the table $\Gamma[\theta] \xrightarrow{u} M[\theta] : a[\theta]$ there exists a derivation $\Gamma \xrightarrow{u} M : a/\theta$. We will keep all the previous inference rules, but keep in mind that we are silently passing around a table $\mathcal{T}$. Any substitution we apply to Γ and a (see for example the $\mathsf{fs} \rightarrow$ rule) will not effect the table. This is important because we do want to have explicit control over the table. The application of inference rules should not have any undesired effects on the table. The main judgments are $\mathcal{T}; \Gamma \xrightarrow{u} M : A/(\theta, \mathcal{T}')$ and $\mathcal{T}; \Gamma \gg A \xrightarrow{f} S : a/(\theta, \mathcal{T}')$.

In addition to the $\mathsf{us_atom}$ inference rule, we will have the rules extend and $\mathsf{retrieve}$. The extend rule adds a subgoal and its answer to the table. When we encounter a new subgoal, we add a new entry with an empty answer list to the table. Once we have proven this subgoal, we add the answer substitution and proof term to its answer list, and we can later use it as a lemma. $\mathsf{retrieve}$ allows us to close a branch by applying a lemma from the table. If we are proving $\Gamma \xrightarrow{u} a$, where Γ and a may contain existential variables and we have a proof for $\Gamma[\theta] \xrightarrow{u} M[\theta] : a[\theta]$ in the table then we can just re-use it by substituting the proof term $M[\theta]$for it. Applying the $\mathsf{retrieve}$ rule corresponds to introducing a cut in the proof using a lemma from the table. However, this cut application

$$\frac{\mathsf{extend}(\mathcal{T}, (\Gamma, x : A, \Gamma') \xrightarrow{u} a) = \mathcal{T}_1 \quad \mathcal{T}_1; (\Gamma, x : A, \Gamma') \gg A \xrightarrow{f} S : a/(\theta, \mathcal{T}_2) \quad \mathsf{insert}(\mathcal{T}_2, (\Gamma, x : A, \Gamma') \xrightarrow{u} a, M, \theta) = \mathcal{T}_3}{\mathcal{T}; (\Gamma, x : A, \Gamma') \xrightarrow{u} x \cdot S : a/(\theta, \mathcal{T}_3)}\ \mathsf{extend} \qquad \frac{\mathsf{retrieve}(\mathcal{T}; \Gamma \xrightarrow{u} a) = (\theta, M)}{\mathcal{T}; \Gamma \xrightarrow{u} M : a/(\theta, \mathcal{T})}\ \mathsf{retrieve}$$

Fig. 6. Memoization extensions

is restricted to using only lemmas that are an instance of the sequent we are trying to prove.

We consider $\Gamma \xrightarrow{u} a$ a variant of $\Gamma' \xrightarrow{u} a'$ if there exists a renaming of the bound and existential variables such that $\Gamma \xrightarrow{u} a$ is equal to a'.

Definition 2 (Variant). *The goal $\Gamma \xrightarrow{u} a$ is a variant of $\Gamma' \xrightarrow{u} a'$ if*
- *there exists a bijection between the free variables in $\Gamma \xrightarrow{u} a$ and $\Gamma' \xrightarrow{u} a'$*
- *there exists a bijection between the bound variables in $\Gamma \xrightarrow{u} a$ and $\Gamma' \xrightarrow{u} a'$*

such that such that $\Gamma \xrightarrow{u} a$ is α-convertible to $\Gamma' \xrightarrow{u} a'$.

Now we can define the three main operations on the table, extending the table, inserting an answer in the table and retrieving an answer from the table.

Definition 3 (Extend). *extend*$(\mathcal{T}, \Gamma \xrightarrow{u} a) = \mathcal{T}'$
Let $\mathcal{T}$ be a table, $\Gamma \xrightarrow{u} a$ be a goal.
- *If there exists a table entry $(\Gamma' \xrightarrow{u} a', \mathcal{A})$ in $\mathcal{T}$ and $\mathcal{A}$ is non-empty such that $\Gamma' \xrightarrow{u} a'$ is a variant of $\Gamma \xrightarrow{u} a$ then return $\mathcal{T}$.*
- *If there exists* **no** *table entry $(\Gamma' \xrightarrow{u} a', \mathcal{A})$ in $\mathcal{T}$ such that $\Gamma' \xrightarrow{u} a'$ is a variant of $\Gamma \xrightarrow{u} a$, then we obtain the extended table $\mathcal{T}'$ by renaming all the existential variables in $\Gamma \xrightarrow{u} a$ and adding the renamed goal to the table $\mathcal{T}$ with an empty solution list.*

By renaming all existential variables before adding a goal to the table, we enforce a clear separation between the table and the goals discovered during the application of inference rules.

Definition 4 (Insert). *insert*$(\mathcal{T}, \Gamma \xrightarrow{u} a, M, \theta) = \mathcal{T}'$
Let $\mathcal{T}$ be a table, $\Gamma \xrightarrow{u} a$ be a goal and θ be a corresponding answer substitution and M the proof term. Let $(\Gamma_i \xrightarrow{u} a_i, \mathcal{A})$ be in the table $\mathcal{T}$ and $\Gamma_i \xrightarrow{u} a_i$ is a variant of $\Gamma \xrightarrow{u} a$. If there exists θ_i in the answer substitution list $\mathcal{A}$, such that $\Gamma_i[\theta_i] \xrightarrow{u} a_i[\theta_i]$ is a variant of $\Gamma[\theta] \xrightarrow{u} a[\theta]$, then we fail otherwise we match $\Gamma_i \xrightarrow{u} a_i$ against $\Gamma[\theta] \xrightarrow{u} a[\theta]$ and obtain the substitution θ' where $\theta' = \sigma \circ \theta$. Then we add $(\theta', M[\sigma])$ to $\mathcal{A}$.

Note that the result of matching $\Gamma_i \xrightarrow{u} a_i$ against $\Gamma[\theta] \xrightarrow{u} a[\theta]$ is a substitution θ' such that $\Gamma_i[\theta'] \xrightarrow{u} a_i[\theta']$ is a variant of $\Gamma[\theta] \xrightarrow{u} a[\theta]$. If we discover a sub-goal a in a context Γ, which is already in the table $\mathcal{T}$ but with an empty answer substitution list, then we have discovered a loop in the computation. No

inference rule is applicable, and therefore computation fails. The definitions of extend and insert also prevent us from inferring the same solution twice. If a sub-goal a is already in the table, but has some answers in the answer list $\mathcal{A}$, then we retrieve the answers. As we might need additional answers for a which are not already in the table yet, we need to still be able to apply extend rule.

Definition 5 (Retrieve). *retrieve*$(\mathcal{T}, \Gamma \xrightarrow{u} a) = (\theta, M)$
Let $\mathcal{T}$ be a table and $\Gamma \xrightarrow{u} a$ be a goal. If there exists a table entry $(\Gamma_i \xrightarrow{u} a_i, \mathcal{A}_i)$ such that $\Gamma_i \xrightarrow{u} a_i$ is variant of $\Gamma \xrightarrow{u} a$ and (θ_i, M) is in $\mathcal{A}_i$ then match $\Gamma \xrightarrow{u} a$ against $\Gamma_i[\theta_i] \xrightarrow{u} a_i[\theta_i]$ to obtain a substitution θ where $\theta_i = \sigma \circ \theta$ and return θ and $M = M_i[\sigma]$.

Theorem 3 (Soundness). *If $\mathcal{D} : \mathcal{T}; \Gamma \xrightarrow{u} A/(\theta, \mathcal{T}')$ then for any substitution σ which grounds $\mathcal{T}; \Gamma \xrightarrow{u} A/(\theta, \mathcal{T}')$ we have $\mathcal{E} : \Gamma \xrightarrow{u} A/\theta \circ \sigma$.*

The proof requires again a generalization to include focused proofs, but is otherwise straightforward.

The presented inference rules leave several choices undetermined. For example, we do not specify the order in which we use program clauses. This choice was already present in the non-tabled system. Similarly, the rules to allow memoization leave open in what order we retrieve answers, when to retrieve answers and when to apply program clauses. Although we used variant checking in the definitions, it is possible to allow subsumption checking. In an actual implementation all these choices need to be resolved. The multi-stage strategy discussed earlier is one possible solution. In this strategy we proceed in lock-steps. First, we apply the extend rule until all clauses from Γ have been tried, and then allow the application of the retrieve rule. The strategy also restricts the retrieve rule, i.e. only answers from previous stages can be retrieved. Alternatively, we could use SCC scheduling (strongly connected components), which allows us to consume answers as soon as they are available [22]. In a real implementation we want to suspend goals and be able to later resume them. For example, if a variant of a previous subgoal with no answers is encountered, then search just fails in the presented inference rules. A different combination of clause application however might lead to success. Storing suspended avoids repeating partial work. After some answers have been generated for the sequent $\Gamma \xrightarrow{u} a$, we awaken the suspended goal and resume computation of the pending sub-goals.

5 Related Work and Conclusion

This proof-theoretic view on computation based on memoization provides a high-level description of a tabled logic programming interpreter and separates logical issues from procedural ones leaving maximum freedom to choose particular control mechanisms. In fact, it is very close to our prototype implementation for *Elf*. So far all descriptions of tabling are highly procedural, either designed as an extension of SLD resolution [23] or to the WAM abstract machine[22]. Certificates, which provide evidence for the existence of a proof, have been added

to tabled logic programming by Roychoudhury [21] and are called justifiers. The relationship between the certificate and SLD resolution is extrinsic rather then intrinsic and needs to be established separately. The proof-theoretical characterization offers a uniform framework for describing and reasoning about program clauses, goals and certificates (proof terms). It seems possible to apply the techniques described to other logic programming languages such as λProlog. Linear logic programming [8,2] has been proposed as an extension of higher-order logic programming to model imperative state changes in a declarative (logical) way. We believe our techniques can be extended to cover this case, but it requires some new considerations. In particular, we plan to investigate the interaction between resource management strategies [3] or constraints [7] with tabling.

With tabled uniform proof search we will find fewer proofs than with uniform proofs. For example in the subtyping example given in Sec. 2 the query sub zero zero has infinitely many proofs under the traditional logic programming interpretation while we find only one proof under the tabled logic programming interpretation. However, we often do not want and need to distinguish between different proofs for a formula A, but only care about the existence of a proof for A together with a proof term. In [17] Pfenning develops a dependent type theory for proof irrelevance and discusses potential applications in the logical framework. This allows us to treat all proofs for A as equal if they produce the same answer substitution. In this setting, it seems possible to show that search based on tabled uniform proofs is also non-deterministically complete, i.e. if computation fails, then there exists no proof.

We have implemented the search strategy based on memoization for *Elf*. It not only allows us to execute more specifications, but also execute implementations more efficiently. Preliminary experiments include type checking for subtyping and intersection types, parsing into higher-order abstract syntax, evaluation based on rewriting. The most pressing issue seems to be to implement indexing data-structures to reduce the overhead involved in managing the table[1].

Acknowledgment

The author gratefully acknowledges numerous discussion with Frank Pfenning and David S.Warren concerning this work. Thanks also for many useful comments from C. Schürmann, R. Harper and K. Watkins.

References

1. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, 2000. 271
2. I. Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, 1996. 284

[1] The code to examples can be found at `http://www.cs.cmu.edu/~bp/tabling`

3. I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, 2000. 284
4. I. Cervesato and F. Pfenning. A linear spine calculus. In *submitted*, 2001. 278
5. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996. 274
6. B. Cui, Y. Dong, X. Du, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1998. 271
7. J. Harland and D. Pym. Resource-distribution via boolean constraints. In *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, 1997. Springer-Verlag LNAI 1249. 284
8. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. 284
9. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992. 278, 279
10. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991. 272, 277
11. G. Nadathur and D. Miller. An overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, 1988. MIT Press. 272
12. G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL01)*, 2001. 272
13. L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986. 272
14. F. Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, 1989. IEEE Computer Society Press. 271
15. F. Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, Netherlands, 1991. 278, 279
16. F. Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically. 274
17. F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th Annual IEEE Symposium on Logic in Computer Science*, Boston, USA, 2001. 284
18. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, 1988. 273
19. F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632. 271
20. B. Pientka. Tabled higher-order logic programming. Thesis proposal, Carnegie Mellon University, 2002. 272, 277
21. A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *International Conference on Principles and Practice of Declarative Programming(PPDP'00)*, pages 178–189, 2000. 284

22. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998. 271, 283
23. H. Tamaki and T. Sato. OLD resolution with tabulation. In E. .Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986. 274, 283
24. R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 2000. 276
25. D. S. Warren. *Programming in tabled logic programming*. draft available from http://www.cs.sunysb.edu/ warren/xsbbook/book.html, 1999. 271

Proving the Equivalence of CLP Programs

Sorin Craciunescu

Inria Rocquencourt, Domaine de Voluceau - B.P. 105
78153 Le Chesnay Cedex, France
Sorin.Craciunescu@inria.fr
http://pauillac.inria.fr/~craciune

Abstract. This paper presents two proof systems for the equivalence of programs. The language concerned is CLP to which the universal quantifier is added (CLP$\forall$). Both systems are based on first order classical logic.
The first uses an induction rule and allows one to prove that the set of finite successes of a program is included in another program's corresponding set. The second uses a coinduction rule for proving the inclusion of the sets of infinite successes which contain the finite successes.
Finally we show that the proof systems are equivalent under some natural conditions.

Keywords: Program equivalence, proof system, induction, coinduction, reactive program, logic programming, constraint, CLP, CLP$\forall$.

1 Introduction

Formal systems for proving properties of programs have been studied for a long time and different approaches have been proposed. In this paper the focus is on proving that the set of successes of one program are included into another program's set. By proving the reversed inclusion also one can prove the equivalence of two programs (eg. specification and implementation). Two notions of success will be defined: finite successes which correspond to the "classical" definition and infinite successes which is appropriate for non-terminating reactive programs. Two proof systems will be given, both based on classical logic. The first uses an induction rule for dealing with recursive predicates and the second the dual coinduction rule. For a mathematical presentation of induction and coinduction see [1]. A restricted version of this work was presented in [4].

We add the universal quantifier to the CLP [9] language. The main motivation is to be able to express the negation of an expression of the form $\exists X.E(X)$ as $\forall X.\overline{E}(X)$. This negation will be needed for proving the equivalence of the two proof systems in section 5. However this extension is useful in itself as it increases the expressive power of the language.

There are several proof systems for programs among which Coq[2], HOL[8], Isabelle[13], ACL2[10], LPTP[15], $FO\lambda^{\Delta N}$[12]. Some of them (Coq, HOL) are based on the theory of types - a system which originates in the attempts of

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 287–301, 2002.
© Springer-Verlag Berlin Heidelberg 2002

formalization of mathematics [3]. A main feature is the fact that proofs can be expressed in the same language as propositions by the Curry-Howard isomorphism. Axioms must be introduced into the system for each new inductive type. The system insures their soundness automatically by syntactic restrictions. The drawback is that one cannot expresses programs directly in the system's language.

Other proof systems do allow that. ACL2 allows reasoning about programs written in the LISP language but doesn't have an unique induction rule. LPTP allows reasoning about "impure" Prolog programs: it has a large set of axioms (¿70) which model the operation of stack-based Prolog interpreter and also induction rules.

The system described in [6] is closest to the ones presented here. It allows reasoning about CLP programs and even more expressive CC [14] programs [5]. It has a small set of rules among which an induction rule close to the one used in section 3. However the rule cannot be applied twice on the same predicate. The system allows proving that a program P satisfies a property ϕ, written as $P\ sat\ \phi$. The program and the property are not expressed in the same language therefore one can prove that a program satisfies (logically implies) a specification but not that is equivalent to it (is implied by it). The system has no notion of negation and cannot reason about reactive programs.

The paper is organized as follows.

Section 2 gives the basic definitions used in the rest of the paper. It introduces the CLP$\forall$ language, its syntax and semantics. In this semantics to a CLP$\forall$ goal we associate the set of its finite successes. Some properties of this set are presented which will be used for the proof of the main theorem of the following section.

Section 3 presents the proof system for finite success inclusion. Then the theorem which states its correctness with respect to the semantics given in section 2 follows.

Section 4 begins by defining the set of infinite successes of a CLP$\forall$ program as the complement of its set of finite failures. The set of infinite successes has the properties presented in section 2 for the finite successes. The finite and infinite successes are shown to be the least and greatest fixed points of the same operator. A proof system for the inclusion of infinite successes is presented. The main theorem of this section states its correctness with respect to the semantics of infinite successes.

2 Definitions – The CLP$\forall$ Language

In this section we present the definitions used in the rest of the paper. The syntax and semantics of the CLP$\forall$ language are given together with some properties which will be needed in the following sections.

In the definitions below we assume implicitly that all symbols can be subscripted.

$FunctN$ is the set of function names denoted by f, g, h.

$PredsN$ is the set of predicate names denoted by p, q, r.

$ConstrN$ is the set of constraint names which contains $true$ and $false$ of arity 0.

$ar : FunctN \cup PredsN \cup ConstrN \rightarrow N$ is the function that gives the arity of a function, predicate or constraint.

$Vars$ is the infinite set of variables denoted by U, V, W, X, Y, Z.

$Terms$ is the set of terms denoted by s, t which is the smallest set satisfying the following properties:

$Terms \supseteq Vars$

$Terms \supseteq \{f \in FunctN \,|\, ar(f) = 0\}$

$Terms \supseteq \{f(t_1, .., t_n) \,|\, f \in FunctN,\, ar(f) = n,\, t_i \in Terms,\, \forall i \in \overline{1, n}\}$

$GTerms$ is the set of ground terms denoted by a, b.

$ConstrA$ is the set of atomic constraints of the form $c(t_1, ..., t_n)$ where c is a constraint name of arity n and $t_1, ..., t_n$ are terms .

$Constraints$ is a set of constraints which are first order logical formulae built from atomic constraints, the logical connectives $\vee, \wedge, \rightarrow, \neg$ and the quantifiers $\forall, \exists$. They will be denoted by c, d, e.

$Subst$ is the set of substitutions which are functions $\tau : Vars \rightarrow Terms$ denoted by τ, π. They must be "without cycles" which means that

$$\tau(Y) = Y \; if \; \exists X.(Y \; appears \; in \; \tau(X))$$

This condition is equivalent to the following property of τ: $\forall X.(\tau(\tau(X)) = \tau(X))$.

τE denotes the expression E in which each free occurence of the variable X is replaced by $\tau(X)$.

We use the upper arrow notation for vectors: $\overrightarrow{X}$ denotes a vector $(X_1, X_2, ..., X_n)$ of (different) variables and $\overrightarrow{t}$ denotes a vector $(t_1, t_2, ..., t_m)$ of terms.

The logic language we consider in this paper is CLP$\forall$ which is the usual CLP augmented with the universal quantifier.

It's syntax is the following:

$$\begin{array}{lll} program & ::= & \epsilon \,|\, definition.program \\ definition & ::= & p(\overrightarrow{X}) : -Body(\overrightarrow{X}) \\ exp & ::= & exp\,,\, exp \,|\, exp\,;\, exp \,|\, \exists X.exp \,|\, \forall X.exp \,|\, tell(c) \,|\, p(\overrightarrow{t}) \end{array}$$

Here ϵ is the void string, c is a constraint name and $Body(\overrightarrow{X})$ is a parametric expression (defined below).

Exp is the set of expressions defined by the rule exp above and denoted by $E, F, G, Body$.

$fv : Exp \rightarrow Vars$ is the function that gives the set of free variables of an expression. It's defined in the usual way.

A parametric expression is an expression E for which some of its free variables, $X_1, ..., X_n$, have been chosen as arguments. Such an expression is denoted by $E(X_1, ..., X_n)$ or $E(\overrightarrow{X})$ and $E(t_1, ..., t_n)$ or $E(\overrightarrow{t})$ denotes the expression E in which X_i was substituted by t_i.

Progs is the set of valid programs defined by the rule *program* above and the following restrictions:

- they can contain at most one definition for each predicate name (of a given arity)
- there are no free variables in the body of any clause except for its parameters.

The notation $E\{X/t\}$ stands for "the expression E where all the free occurrences of X have been replaced by t".

$\vdash_D$ is any binary relation on *Constraints* which satisfies the following properties:

- for all $c \in Constraints$ we have $false \vdash_D c$, $c \vdash_D true$ and $c \vdash_D c$.
- if $c_1 \vdash_D c_2$ and $c_2 \vdash_D c_3$ then $c_1 \vdash_D c_3$ (transitivity of $\vdash_D$).
- if $c_1 \vdash_D c_2$ then $\forall t \in Terms$, $\forall X \in Vars$ we have $c_1\{X/t\} \vdash_D c_2\{X/t\}$.

Remark: the first-order entailment relation $\bullet \vdash_{FO} \bullet$ between constraints defined in the usual way satisfies all the properties required for $\vdash_D$.

$Constraints^*$ is the set defined by $\bigcup_{n \in N^*} Constraints^n$ where $Constraints^n = Constraints \times ... \times Constraints$ ($n > 0$ times).

$\vdash_{DM}$ is a binary relation on $Constraints^*$ for which we will use the following notation: if $x = (c_1, .., c_n) \in Constraints^n$ and $y = (d_1, .., d_m) \in Constraints^m$ then we will write $x \vdash_{DM} y$ as $c_1, .., c_2 \vdash_{DM} d_1; ..; d_m$. The relation is uniquely defined by the following properties:

- $c \vdash_{DM} d$ iff $c \vdash_D d$.
- $c_1, .., c_n \vdash_{DM} d_1; ..; d_m$ iff for all $e \in Constraints$ such that $\forall i \in \overline{1..n}$, $e \vdash_D c_i$ there exists $j \in \overline{1..m}$ such that $e \vdash_D d_j$.

We now extend the notation $\vdash_{DM}$ to stand for two unary relations on $Constraints^n$: $\bullet \vdash_{DM}$ and $\vdash_{DM} \bullet$ defined as follows:

- $c_1, .., c_n \vdash_{DM}$ iff $c_1, .., c_n \vdash_{DM} false$
- $\vdash_{DM} d_1; ..; d_m$ iff $true \vdash_{DM} d_1; ..; d_m$

Together, $ConstrN$, $\vdash_D$ and $\vdash_{DM}$ form the constraint domain D which is a parameter of the CLP∀ language.

Remark: the relation $\vdash_{DM}$ doesn't have to be totally computable for the proof systems defined in the following sections to be correct. However, being only partially computable might reduce the number of proofs that can be built in those systems.

The relation $\vdash_{DM}$ is easily computable if the constraint domain has the *Independence of Negated Constraints* property (INC) [11]. An example of such domain is the Herbrand domain (H) where the only constraint is the unifiability of terms ("=").

The semantics of the CLP∀ language is given by a sequent calculus:

$$P, c \vdash_s E$$

which should be interpreted as "c is a success (or solution) of E in the context of the program P". The subscript "s" stands for "success".

The rules of the calculus are:

$$\frac{}{P,\, c \vdash_s E}(false) \qquad \frac{}{P,\, c_1 \vdash_s tell(c_2)}(tell)$$

with the side conditions: $c \vdash_D false$ for $(false)$ and $c_1 \vdash_D c_2$ for $(tell)$

$$\frac{P,\, c \vdash_s E_1 \quad P,\, c \vdash_s E_2}{P,\, c \vdash_s (E_1, E_2)}(,) \qquad \frac{P,\, c \vdash_s E_1}{P,\, c \vdash_s (E_1; E_2)}(;1) \qquad \frac{P,\, c \vdash_s E_2}{P,\, c \vdash_s (E_1; E_2)}(;2)$$

$$\frac{P,\, c \vdash_s Body(\overrightarrow{t})}{P,\, c \vdash_s p(\overrightarrow{t})}(def.)$$

where $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$

$$\frac{P,\, c \vdash_s E(t)}{P,\, c \vdash_s \exists X.E(X)}(\exists) \qquad \frac{P,\, c \vdash_s E(Y)}{P,\, c \vdash_s \forall X.E(X)}(\forall)$$

$(\forall)$ having the side condition: $Y \notin fv(c) \cup fv(E(X))$.

$Succ : Progs \times Exp \to \wp(Constraints)$ is the function that gives the set of successes of an expression E in the context of a program P:

$$Succ(P, E) = \{c \in Constraints \mid (P,\, c \vdash_s E)\}$$

Example: the following CLP$\forall(H)$ program called NI:

```
nat(X):-tell(X=0)
      ; ∃Y.(tell(X=s(Y)), nat(Y)).
int(X):-tell(X=0)
      ; ∃Y.(tell(X=s(Y)), int(Y))
      ; ∃Z.(tell(X=p(Z)), int(Z)).
```

defines two predicates, *nat* and *int* which have the following sets of successes:

$Succ(NI, nat(X)) = \{X = 0, X = s(0), ...\}$

$Succ(NI, int(X)) = \{X = 0, X = s(0), ..., X = p(0), X = p(s(0)), ...\}$

In the previous program we have written $t_1 = t_2$ instead of $tell(t_1 = t_2)$.

Lemma 2.1: For each CLP$\forall$ program P, predicate $p(\overrightarrow{X})$ defined in P by $p(\overrightarrow{X}) : -Body(\overrightarrow{X})$, expressions E, E_1, E_2 and term t the following properties hold:

1. $Succ(P, tell(false)) \subseteq Succ(P, E)$.
2. $Succ(P, p(\overrightarrow{t})) = Succ(P, Body(\overrightarrow{t}))$.
3. $Succ(P, (E_1, E_2)) = Succ(P, E_1) \cap Succ(P, E_2)$.
4. $Succ(P, (E_1; E_2)) = Succ(P, E_1) \cup Succ(P, E_2)$.
5. $Succ(P, (\exists X.E(X))) = \bigcup_{t \in Terms} Succ(P, E(t))$.
6. $Succ(P, (\forall X.E(X))) = \bigcap_{t \in Terms} Succ(P, E(t))$.

7. if $Succ(P, E_1) \subseteq Succ(P, E_2)$ then $\forall t \in Terms$, $Succ(P, E_1\{X/t\}) \subseteq Succ(P, E_2\{X/t\})$.
8. if $Succ(P, E_1) \subseteq Succ(P, E_2)$ then $\forall \sigma \in Subst$, $Succ(P, \sigma E_1) \subseteq Succ(P, \sigma E_2)$.

Proof: by induction over the structure of the derivation tree for the expressions involved.

We define now the approximations of a predicate $p(\overrightarrow{X})$ which will be used in the proof of the upcoming theorem 3.1.

If $p(\overrightarrow{X}) : -Body(\overrightarrow{X})$ is a definition in the program P then p may call itself directly if $p(\overrightarrow{t})$ appears in $Body(\overrightarrow{X})$ or indirectly if p calls q_1 which calls q_2,...,which calls $q_n$which calls p. In the second case we can obtain an equivalent definition, $p_d(\overrightarrow{X}) : -Body_d(\overrightarrow{X})$ in which p_d doesn't call itself indirectly by unfolding the definitions of $q_1, .., q_n$ in the body of p. We have of course that

$$Succ(P, p(\overrightarrow{t})) = Succ(P, p_d(\overrightarrow{t}))$$

$Apx(P, p)$ is the program obtained by adding to P the following definitions (we suppose that $p^n, n \in N$ are fresh predicate names):

$p^0(\overrightarrow{X}) : -tell(false)$.

$p^1(\overrightarrow{X}) : -Body_d(\overrightarrow{X})\{p/p^0\}$.

....

$p^{n+1}(\overrightarrow{X}) : -Body_d(\overrightarrow{X})\{p/p^n\}$.

.... for every $n \in N$. Here $Body_d(\overrightarrow{X})\{p/p^n\}$ denotes the expression $Body_d(\overrightarrow{X})$ where all the calls to p have been replaced with calls to p^n. The predicates $p^0, .., p^n, ..$ are called "the (finite) approximations of p".

Example: for the previous CLP∀ program "NI" we can construct $Apx(NI, nat)$ by adding to $NI$the following definitions:

```
nat⁰(X):-tell(false).
nat¹(X):-tell(X=0) ; ∃Y.(tell(X=s(Y)), nat⁰(Y)).
nat²(X):-tell(X=0) ; ∃Y.(tell(X=s(Y)), nat¹(Y)). ...
```

Lemma 2.2: For each program P and predicate p defined in P the following property holds:

$$Succ(P, p(\overrightarrow{t})) = \bigcup_{n \in N} Succ(Apx(P, p), p^n(\overrightarrow{t}))$$

Let P be a CLP program according to the syntax given in [11] and $p(\overrightarrow{X})$ a predicate defined in P. We define $Trans(P)$, the translation of P into the CLP∀ language as the program which contains the following definition of $p(\overrightarrow{X})$:

$p(\overrightarrow{X}) : -Body_1(\overrightarrow{X}) ; ..; Body_n(\overrightarrow{X})$

if P contains the clauses:

$p(\overrightarrow{X}) : -S_1$.

...

$p(\overrightarrow{X}) : -S_n$.

Here $Body_i(\overrightarrow{X}) = Trans(S_i)$ where $Trans(S_i)$is the translations of the goal S_i: if S_i is a sequence containing the atomic constraints $c_1, .., c_n$ and the calls $p_1(\overrightarrow{X_1}), .., p_n(\overrightarrow{X_n})$ then $Body_i(\overrightarrow{X}) = \exists Y_1.\exists Y_2 ... \exists Y_m.(Body_i^{'})$. Here $Body_i^{'}$ is the corresponding conjunction where c_j is replaced by $tell(c_j)$ and $Y_1, .., Y_m$ are all the variables that appear in S_i but not in the vector $\overrightarrow{X}$ (the "local variables" of the clause).

Proposition 2.3: If P is a CLP program, S a CLP goal and c a constraint such that $< S, true > \rightarrow^* < \Box, c >$ then $\forall d \in Constraints$ such that $d \vdash_D c$ we have $d \in Succ(Trans(P), Trans(S))$.

Remark: the inverse translation - from a CLP$\forall$ program without $\forall$ into a CLP program - is also possible (and straightforward).

3 A Proof System for Success Equivalence

In this section we present an induction-based proof system for the inclusion of successes and its proof of soundness.

We want to prove that for a program P and two expressions (goals) E, F we have:

$$Succ(P, E) \subseteq Succ(P, F)$$

We will use for that purpose a proof system expressed by means of a sequent calculus of the form:

$$P, \Sigma \vdash_{si} E \rhd F$$

where Σ is a multiset of elements of the form $E_1 \rhd F_1$. They are called "hypotheses" and are denoted by the letter H (possibly subscripted). The subscript "si" stands for "success inclusion".

The meaning of the sequent above is "in the context of the program P and hypotheses Σ the successes of E are included in the successes of F". The meaning of the hypothesis $E_1 \rhd F_1$ is "in the context of the program P the successes of E_1 are included in the successes of F_1".

The elements of the multiset Σ will separated by ",". Therefore $[(E_1 \rhd F_1), \Sigma]$ denotes the multiset composed by the hypotheses $E_1 \rhd F_1$ and the elements of the multiset Σ.

To make the rules clearer we will use Γ, Δ as symbols for CLP$\forall$ expressions.

The proof system is given by the following rules:

$$\frac{}{P,\ \Sigma \vdash_{si} E, \Gamma \rhd E; \Delta}(id.)$$

$$\frac{P,\ \Sigma \vdash_{si} E_n, E_1, .., E_{n-1} \rhd \Delta}{P,\ \Sigma \vdash_{si} E_1, .., E_n \rhd \Delta}(comm.\ L) \quad \frac{P,\ \Sigma \vdash_{si} \Gamma \rhd E_n; E_1; ..; E_{n-1}}{P,\ \Sigma \vdash_{si} \Gamma \rhd E_1; ..; E_n}(comm.\ R)$$

$$\frac{P,\ \Sigma \vdash_{si} E_1, E_2, \Gamma \rhd \Delta}{P,\ \Sigma \vdash_{si} (E_1, E_2), \Gamma \rhd \Delta}(, L) \quad \frac{P,\ \Sigma \vdash_{si} \Gamma \rhd E_1; \Delta \quad P,\ \Sigma \vdash_{si} \Gamma \rhd E_2; \Delta}{P,\ \Sigma \vdash_{si} \Gamma \rhd (E_1, E_2); \Delta}(, R)$$

$$\frac{P,\ \Sigma \vdash_{si} E_1, \Gamma \triangleright \Delta \quad P,\ \Sigma \vdash_{si} E_2, \Gamma \triangleright \Delta}{P,\ \Sigma \vdash_{si} (E_1; E_2), \Gamma \triangleright \Delta}(;L) \quad \frac{P,\ \Sigma \vdash_{si} \Gamma \triangleright E_1; E_2; \Delta}{P,\ \Sigma \vdash_{si} \Gamma \triangleright (E_1; E_2); \Delta}(;R)$$

$$\frac{}{P,\ \Sigma \vdash_{si} tell(c_1), .., tell(c_n), \Gamma \triangleright tell(d_1); ..; tell(d_m); \Delta}(tell)$$

if $c_1, .., c_n \vdash_{DM} d_1; ..; d_m$ and $n > 0$ or $m > 0$.

$$\frac{P,\ \Sigma \vdash_{si} E(Y), \Gamma \triangleright \Delta}{P,\ \Sigma \vdash_{si} \exists X.E(X), \Gamma \triangleright \Delta}(\exists L) \quad \frac{P,\ \Sigma \vdash_{si} \Gamma \triangleright E(t); \Delta}{P,\ \Sigma \vdash_{si} \Gamma \triangleright \exists X.E(X); \Delta}(\exists R)$$

$(\exists L)$ having the side condition: $Y \notin fv((E(X), \Gamma)) \cup fv(\Delta)$

$$\frac{P,\ \Sigma \vdash_{si} E(t), \Gamma \triangleright \Delta}{P,\ \Sigma \vdash_{si} \forall X.E(X), \Gamma \triangleright \Delta}(\forall L) \quad \frac{P,\ \Sigma \vdash_{si} \Gamma \triangleright E(Y); \Delta}{P,\ \Sigma \vdash_{si} \Gamma \triangleright \forall X.E(X); \Delta}(\forall R)$$

$(\forall R)$ having the side condition: $Y \notin fv((E(X); \Delta)) \cup fv(\Gamma)$

$$\frac{P,\ \Sigma \vdash_{si} Body(\overrightarrow{t}), \Gamma \triangleright \Delta}{P,\ \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma \triangleright \Delta}(def.\,L) \quad \frac{P,\ \Sigma \vdash_{si} \Gamma \triangleright Body(\overrightarrow{t}); \Delta}{P,\ \Sigma \vdash_{si} \Gamma \triangleright p(\overrightarrow{t}); \Delta}(def.\,R)$$

$(def.\,L)$ and $(def.\,R)$ having the side condition: $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

$$\frac{P,\ \Sigma \vdash_{si} E' \triangleright F'}{P,\ \Sigma \vdash_{si} E \triangleright F}(gen.)$$

if $\exists \tau$ such that $\tau E' = E$, $\tau F' = F$.

$$\frac{P,\ [(F \triangleright G), \Sigma] \vdash_{si} E', \Gamma \triangleright \Delta}{P,\ [(F \triangleright G), \Sigma] \vdash_{si} E, \Gamma \triangleright \Delta}(hyp.\,L)$$

if $\exists \tau$ s.t. $\tau F = E$, $\tau G = E'$.

$$\frac{P,\ [((p(\overrightarrow{t}), \Gamma) \triangleright \Delta), \Sigma] \vdash_{si} Body(\overrightarrow{t}), \Gamma \triangleright \Delta}{P,\ \Sigma \vdash_{si} p(\overrightarrow{t}), \Gamma \triangleright \Delta}(ind.)$$

$(ind.)$ having the side condition: $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

Theorem 3.1.: Soundness of the proof system $\vdash_{si}$.
If the sequent

$$P,\ \Sigma \vdash_{si} E \triangleright F$$

where $\Sigma = [E_1 \triangleright F_1, .., E_p \triangleright F_p]$ is provable and

$$Succ(P, E_i) \subseteq Succ(P, F_i),\ \forall i \in \overline{1..p}$$

then

$$Succ(P, E) \subseteq Succ(P, F)$$

Proof: by induction on the structure of the proof tree of the sequent $P,\ \Sigma \vdash_{si} E \triangleright F$. For details see [4] or the author's web page: "http://pauillac.inria.fr/~craciune".

Corollary 3.2.: If the following sequent is provable

$$P,\ [] \vdash_{si} E \triangleright F$$

where [] denotes the void multiset (of hypotheses) then

$$Succ(P, E) \subseteq Succ(P, F)$$

Example:
for the "NI" program given above we want to prove that the successes of $nat(X)$ are included in those of $int(X)$.
We prove the following sequent

$$NI,\ [] \vdash_{si} nat(X) \triangleright int(X)$$

in the $\vdash_{si}$ proof system:

$$\dfrac{\dfrac{\dfrac{}{NI, [H_{ni}] \vdash_{si} tell(X=0) \triangleright tell(X=0)}(def.R,\ id.) \quad \dfrac{\Omega}{NI, [H_{ni}] \vdash_{si} \exists Y.(\ldots) \triangleright int(X)}(def.R)}{NI, [nat(X) \triangleright int(X)] \vdash_{si} tell(X=0); \exists Y.(\ldots) \triangleright int(X)}(;L)}{NI, [] \vdash_{si} nat(X) \triangleright int(X)}(ind.)main$$

where Ω is the following subtree (the subtree Θ was omitted):

$$\dfrac{\Theta \qquad \dfrac{}{NI, [H_{ni}] \vdash_{si} nat(X), \ldots \triangleright int(X); \ldots}(hyp.\ L,\ id.)}{NI,\ [H_{ni}] \vdash_{si} \exists Y.(tell(X = Y), nat(Y)) \triangleright tell(X = 0); \exists Y.(tell(X = s(Y)), int(Y)); \ldots}(\ldots)$$

Here H_{ni} is the hypothesis $nat(X) \triangleright int(X)$.
By the corollary 3.2. we have that

$$Succ(NI, nat(X)) \subseteq Succ(NI, int(X))$$

4 A Proof System for Finite Failure Equivalence

In this section we present a proof system - dual to the one in the previous section - for proving the inclusion of finite failures of two expressions (goals).

We first need to define the set of finite failures of an expression. We use a sequent calculus of the form:

$$P,\ c \vdash_{ff} E$$

which is to interpreted as "c is a finite failure of the expression (goal) E in the context of the program P". The subscript "ff" stands for "finite failure". Informally, the finite failures of a an expression E are those constaints c such

that the breadth-first interpreter of the CLP∀ language constructed from the rules of $\vdash_s$ calculus fails finitely when searching a derivation for $P,\, c \vdash_s E$.

The rules of the calculus are the following:

$$\frac{}{P, c_1 \vdash_{ff} tell(c_2)}(tell)$$

$(tell)$ having the side condition: $c_1 \nvdash_D c_2$.

$$\frac{P,\, c \vdash_{ff} E_1}{P,\, c \vdash_{ff} (E_1, E_2)}(,1) \quad \frac{P,\, c \vdash_{ff} E_2}{P,\, c \vdash_{ff} (E_1, E_2)}(,2) \quad \frac{P,\, c \vdash_{ff} E_1 \quad P,\, c \vdash_{ff} E_2}{P,\, c \vdash_s (E_1; E_2)}(;)$$

$$\frac{P,\, c \vdash_{ff} Body(\overrightarrow{t})}{P,\, c \vdash_{ff} p(\overrightarrow{t})}(def.)$$

where $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$

$$\frac{P,\, c \vdash_{ff} E(Y)}{P,\, c \vdash_{ff} \exists X.E(X)}(\exists) \qquad \frac{P,\, c \vdash_{ff} E(t)}{P,\, c \vdash_{ff} \forall X.E(X)}(\forall)$$

$(\exists)$ having the side condition: $Y \notin fv(c) \cup fv(E(X))$.

$FFail : Progs \times Exp \to Constraints$ is the function that gives the set of finite failures of an expression E in the context of a program P:

$$FFail(P, E) = \{c \in Constraints \,|\, (P,\, c \vdash_{ff} E)\}$$

$ISucc : Progs \times Exp \to Constraints$ is the function that gives the set of infinite successes of an expression E in the context of a program P:

$$ISucc(P, E) = Constraints - FFail(P, E)$$

Remark: $ISucc$ has all the properties of $Succ$ given in lemma 2.1..

The following definition is of interest for the semantics of CLP∀:

$IE : Progs \times (Exp \to \wp(Constraints)) \to (Exp \to \wp(Constraints))$ is an operator defined recursively as follows: if $A : Exp \to \wp(Constraints)$ and $p(\overrightarrow{X}) : -Body(\overrightarrow{X})$ is a predicate definition then

$IE(P, A)(tell(c)) = \{d \in Constraints \,|\, d \vdash_D c\}$

$IE(P, A)((E_1, E_2)) = IE(P, A)(E_1) \cap IE(P, A)(E_2)$

$IE(P, A)((E_1; E_2)) = IE(P, A)(E_1) \cup IE(P, A)(E_2)$

$IE(P, A)(\exists X.E_1(X)) = \bigcup_{t \in Terms} IE(P, A)(E_1(t))$

$IE(P, A)(\forall X.E_1(X)) = \bigcap_{t \in Terms} IE(P, A)(E_1(t))$

$IE(P, A)(p(\overrightarrow{t})) = IE'(P, A)(Body(\overrightarrow{t}))$

where IE' is an operator defined identically to IE except that $IE'(P, A)(p(\overrightarrow{t}))$ $= A(p(\overrightarrow{t}))$.

The infinite approximations of a predicate $p(\overrightarrow{t})$ are similar to the finite approximations defined previously. They will be denoted by $p^{i,n}(\overrightarrow{t})$ with $n \in N$.

$IApx(P, p)$ is defined accordingly:

$p^{i,0}(\overrightarrow{X}) : -tell(true).$
$p^{i,1}(\overrightarrow{X}) : -Body_d(\overrightarrow{X})\{p/p^{i,0}\}.$
....
$p^{i,n+1}(\overrightarrow{X}) : -Body_d(\overrightarrow{X})\{p/p^{i,n}\}.$
....

We denote by $Succ(P) : Progs \rightarrow (Exp \rightarrow \wp(Constraints))$ the function such that $Succ(P)(E) = Succ(P, E)$. Similarly $ISucc(P)(E) = ISucc(P, E)$.

Lemma 4.1: The following properties hold for each CLP$\forall$ program P and predicate $p(\overrightarrow{X})$:

1. $ISucc(P, p(\overrightarrow{t})) = \bigcap_{n \in N} ISucc(IApx(P, p), p^{i,n}(\overrightarrow{t}))$
2. $Succ(P) = \mu A.IE(P, A)$
3. $ISucc(P) = \nu A.IE(P, A)$

where μ, ν are the usual operators for the least and greatest fixed points.

We are now ready to define a sequent calculus for proving the inclusion of the sets of finite failures of two expressions. A sequent of the form:

$$P,\ \Sigma \vdash_{ffi} E \triangleright F$$

is to be interpreted as "in the context of the program P and hypotheses Σ the set of finite failures of E includes the finite failures of F".

All the rules of the proof system $\vdash_{ff}$ are identical to those of the system $\vdash_s$ except the rules $(hyp.\ L)$ and $(ind.)$ which are replaced by the following rules:

$$\frac{P,\ [(F \triangleright G), \Sigma] \vdash_{ffi} \Gamma \triangleright E'; \Delta}{P,\ [(F \triangleright G), \Sigma] \vdash_{ffi} \Gamma \triangleright E; \Delta}(hyp.\ R)$$

with the side condition $\exists \tau$ such that $\tau G = E$, $\tau F = E'$.

$$\frac{P,\ [(\Gamma \triangleright ((p(\overrightarrow{t}); \Delta)), \Sigma] \vdash_{ffi} \Gamma \triangleright Body(\overrightarrow{t}); \Delta}{P,\ \Sigma \vdash_{ffi} \Gamma \triangleright p(\overrightarrow{t}); \Delta}(coind.)$$

with the side condition $(p(\overrightarrow{X}) : -Body(\overrightarrow{X})) \in P$.

Theorem 4.2. Soundness of the proof system $\vdash_{ffi}$.
If the sequent

$$P,\ \Sigma \vdash_{ffi} E \triangleright F$$

where $\Sigma = [H_{11} \triangleright H_{12}, .., H_{p1} \triangleright H_{p2}]$ is provable and

$$ISucc(P, H_{i1}) \subseteq ISucc(P, H_{i2}),\ \forall i \in \overline{1..p}$$

then

$$ISucc(P, E) \subseteq ISucc(P, F)$$

Proof: dual to the one of the theorem 3.1..

Example: consider the following program called *Lst* containing two non-terminating predicates:

```
list0(L):-
    ∃L1.(tell(L=[0 | L1]), list0(L1)).
list01(L):-
    ∃L1.((tell(L=[0 | L1]) ; tell(L=[1 | L1])), list01(L1)).
```

The predicate *list0* "checks" that its argument is an infinite list of 0 in the sense that it fails for any other value. We can say that its only success is the infinite list of 0 elements.

The predicate *list01* doesn't terminate when its argument is an infinite lists composed only of 0 and 1 and fails for any other value. We want to prove that the finite failures of *list1* are included into those of *list0* which means that *list01* can do any (infinite) derivation that *list0* can.

Here is the proof:

$$\frac{\dfrac{\overline{Lst, [H]\vdash_{ffi} tell(L=[0 \mid L2]), list0(L2)\triangleright tell(L=[0 \mid L2])}\quad \Omega}{\dfrac{Lst, [list0(L)\triangleright list01(L)]\vdash_{ffi} \exists L1.(\ldots)\triangleright\exists L1.((tell(L=[0 \mid L1]);\ldots)}{Lst, []\vdash_{ffi} list0(L)\triangleright list01(L)}(coind.)}(\exists L, \exists R, ; R)$$

where Ω if the following proof-tree:

$$\frac{}{Lst,\ [H] \vdash_{ffi} tell(L = [0 \mid L2]), list0(L2) \triangleright tell(L = [0 \mid L2]), list01(L2)}(hyp.\ R,\ id)$$

where H is the hypothesis $list0(L) \triangleright list01(L)$. Once more, in order to prove $list0(L) \triangleright list01(L)$ we have to prove $list0(L) \triangleright list01(L)$which we can do using the hypothesis introduced by (*coind.*).

Remark: we can also prove

$$Lst\ [] \vdash_{si} list0(L) \triangleright list01(L)$$

but this is not interesting as we would prove $Succ(Lst, list0(L)) \subseteq Succ(Lst, list01(L))$ and both sets of successes are void.

5 Equivalence of the Two Proof Systems

In this section we prove the equivalence of the two proof systems by using negation. For each program P we define its negation $\overline{P}$ and we prove that the finite successes (finite failures) of an expression E in the context of P are the finite failures (successes) of $\overline{E}$ in the context of $\overline{P}$.

In this section we suppose that for each constraint c there exists a constraint $\overline{c}$ (of the same arity) such that $\overline{\overline{c}} = c$ and if $d \vdash_D c$ and $d \vdash_D \overline{c}$ then $d \vdash_D false$.

We define the negation $\overline{P}$ of a program P as a program which contains the predicate non_p iff P contains the predicate p (we suppose there is no name-clash). If p is defined by:

$$p(\overrightarrow{X}) : -Body(\overrightarrow{X})$$

then non_p is defined by:

$$non_p(\overrightarrow{X}) : -\overline{Body(\overrightarrow{X})}$$

The negation $\overline{E}$ of an expression E is defined as follows:

$$\overline{A, B} = \overline{A}; \overline{B} \qquad \overline{A; B} = \overline{A}, \overline{B}$$

$$\overline{tell(c(\overrightarrow{t}))} = tell(\overline{c}(\overrightarrow{t}))$$

$$\overline{\forall X.E(X)} = \exists X.\overline{E}(X) \qquad \overline{\exists X.E(X)} = \forall X.\overline{E}(X)$$

$$\overline{p(\overrightarrow{t})} = non_p(\overrightarrow{t})$$

We define the negation $\overline{\Sigma}$ of a multiset Σ of hypotheses $E_i \triangleright F_i$ as the multiset containing the elements $\overline{F_i} \triangleright \overline{E_i}$.

Remark: in the previous definition we suppose that all the variables in $Body(\overrightarrow{X})$ are either quantified or appear in $p(\overrightarrow{X})$. If this is not the case the remaining variables are treated as they were existentially quantified at the scope of the clause body.

$False$ is the set of constraints defined by $\{d \in Constraints \mid d \vdash_D false\}$.

Lemma 5.1. The following properties hold:

- $\overline{\overline{E}} = E$
- $Succ(P, E) = FFail(\overline{P}, \overline{E}) \cup False$

Proof: induction on the structure of the expression E and the proof tree respectively.

Theorem 5.2. Equivalence of proof systems $\vdash_{si}$ and $\vdash_{ffi}$.

$$P, \Sigma \vdash_{si} E \triangleright F$$

is provable iff

$$\overline{P}, \overline{\Sigma} \vdash_{ffi} \overline{F} \triangleright \overline{E}$$

is provable.

Proof: induction on the structure of the proof tree, the two proof systems being symmetrical with respect to negation and left-right inversion.

Remark: lemma 5.1. and theorem 5.2. show that "normal" (non-reactive) and reactive CLP$\forall$ programs are essentially equivalent with respect to successes and one can reason about them using the same proof system if some natural conditions (existence of negated constraints) are met.

6 Conclusion

We have presented two systems for proving the equivalence of programs in the CLP language with the universal quantifier (CLP∀). One uses an induction rule for proving the inclusion of finite successes and the other uses coinduction for proving the inclusion of infinite successes (for reactive programs). The systems are based on classical logic, have a small set of rules, allow reasoning directly on programs without the need for additional axioms.

A basic implementation of a proof checker exists and a more advanced one (proof assistant) is currently in making. The induction/coinduction rules are well suited for automatic proof search as they provide the induction hypotheses directly.

An interesting perspective is extending the proof systems to allow reasoning on more expressive languages like CC [14] or its non-monotonic linear-logic based extension LCC [7].

I would like to thank Dale Miller, Slim Abdennadher and especially Francois Fages for interesting discussions and suggestions about this work.

References

1. Aczel, P.: An Introduction to Inductive Definitions. Barwise K., Ed.. Handbook of Mathematical Logic. North Holland, 1977. 287
2. Barras, B. et al.: The Coq Proof Assistant. Reference Manual. http://coq.inria.fr/doc/main.html. 287
3. Church, A.: A Formulation of the Simple Theory of Types. Journal of Symbolic Logic, 5:56-68, 1940. 288
4. Craciunescu, S.: Preuves de Programmes Logiques par Induction et Coinduction (Proofs of Logic Programs by Induction and Coinduction). In Actes de 10e Journees Francophones de Programmation en Logique et de Programmation par Contraintes, (JFPLC'2001), Paris, France. 287, 295
5. de Boer, F. S., Gabbrielli, M., Marchiori, E., Palamidessi, C: Proving Concurrent Constraint Programs Correct. ACM-TOPLAS 19(5):685-725, 1997. 288
6. de Boer, F. S., Palamidessi, C.: A process algebra for concurrent constraint programming. In Proc. Joint Int'l Conf. and Symp. on Logic Programming, pages 463-477, The MIT Press, 1992. 288
7. Fages, F., Ruet, P., Soliman, S.: Linear concurrent constraint programming: operational and phase semantics. Information and Computation no. 164, 2001. 300
8. Gordon, J. C., Melham, T. F. (eds.): Introduction to HOL. Cambridge University Press 1993 ISBN 0-521-441897. 287
9. Jaffar, J., Lassez, J.-L. Constraint Logic Programming. Proceedings of Principles of Programming Languages, 1987, Munich, 111-119. 287
10. Kaufmann, M., Manolios, P., Moore, J. S.: Computer-Aided Reasoning: An Approach Kluwer Academic Publishers, June, 2000. (ISBN 0-7923-7744-3). 287
11. Maher, M. J.: Adding Constraints to Logic-based Formalisms, in: The Logic Programming Paradigm: a 25 Years Perspective, K. R. Apt, V. Marek, M. Truszczynski and D. S. Warren (Eds.), Springer-Verlag, Artificial Intelligence Series, 313-331, 1999. 290, 292

12. McDowell, R., Miller, D.: Cut-Elimination for a Logic with Definitions and Induction. Theoretical Computer Science, 232: 91 - 119,2000. 287
13. Paulson, L. C.: The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993. ftp://ftp.cl.cam.ac.uk/ml/ref.dvi.gz. 287
14. Saraswat, V., Rinard, M.: Concurrent constraint programming. ACM Symposium on Principles of Programming Languages 1990, San Francisco. 288, 300
15. Stärk, R. F.: The theoretical foundations of LPTP (a logic program theorem prover). Journal of Logic Programming, 36(3):241-269, 1998. 287

A Purely Logical Account of Sequentiality in Proof Search

Paola Bruscoli
Technische Universität Dresden
Fakultät Informatik - 01062 Dresden - Germany
Paola.Bruscoli@Inf.TU-Dresden.DE

Abstract A strict correspondence between the proof-search space of a logical formal system and computations in a simple process algebra is established. Sequential composition in the process algebra corresponds to a logical relation in the formal system in this sense our approach is purely logical, no axioms or encodings are involved. The process algebra is a minimal restriction of CCS to parallel and sequential composition; the logical system is a minimal extension of multiplicative linear logic. This way we get the first purely logical account of sequentiality in proof search. Since we restrict attention to a small but meaningful fragment, which is then of very broad interest, our techniques should become a common basis for several possible extensions. In particular, we argue about this work being the first step in a two-step research for capturing most of CCS in a purely logical fashion.

1 Introduction

One of the main motivations of logic programming is the idea of using a high level, logical specification of an algorithm, which abstracts away from many details related to its execution. As Miller pointed out, logical operators can be interpreted as high level search instructions, and the sequent calculus can be used to give a very clear and simple account of logic programming [13].

In traditional logic programming, one is mainly interested in the result of a computation, and computing is essentially the exploration of a search space. Recently, Miller's methods have been extended to so-called resource-conscious logics, like linear logic [4, 12], and researchers designed several languages based on them [2, 10, 12]. These logics allow to deal directly with notions of *resources*, *messages*, *processes*, and so on; in other words, it is possible to give a proof-theoretical account of concurrent computations, in the logic programming spirit.

A concurrent computation is not as much about getting a result, as it is about establishing certain communication patterns, protocols, and the like. Hence we might wonder to which extent logic can be useful in the specification of concurrent programs. Differently stated, if concurrent programs are essentially protocols, subject mainly to an operational view of computation, can logic contribute to their design? We are not concerned here about the use of logics to prove properties of programs, like, say, Hennessy-Milner logic for CCS. We want to use logic in the design of languages for concurrent computation, in order to obtain some useful *inherent* properties, at the object level, so to speak.

In this paper I will present a very simple process algebra and I will argue about its proof-theoretical understanding in terms of proof-search. We will work within the calculus of structures [7], which is a recent generalisation of the sequent

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 302-316, 2002.
© Springer-Verlag Berlin Heidelberg 2002

calculus [3]. Guglielmi and Tiu showed how it is possible to design, in the calculus of structures, a simple logical system which possesses a self-dual non-commutative operator [7], and how this system can not be defined in the sequent calculus [16]. This non-commutative operator, called *seq*, has a resemblance to the prefix combinator of CCS [14]; it is a form of sequential composition, similar to other sequential constructs in other languages. (We should not forget that sequential composition has a longer history than parallel forms of composition, which more naturally correspond to the usual commutative logical operators.)

We will consider the simplest system containing seq, called system SBV: it is not very expressive (it is decidable), but contains the hard part of our problem. Beyond seq, SBV has two commutative logical operators, corresponding to linear logic's *par* and *times*. Several steps have to be made before a real language can be designed starting from SBV:

1 The correspondence between seq and a form of sequentiality studied independently must be established.

2 The search space for proofs must be narrowed enough to get the desired behaviour at run-time.

3 SBV must be extended to a Turing-equivalent fragment and the two properties above must be preserved.

In this paper we will deal with 1 and, partially, with 2, and I will argue about the possibility of completing the program in future work. Let us see in more detail what the three issues above are about.

Point 1: I believe that logic, in the sense of the formal study of language, should give an account of *existing* languages (as opposed to the creation of new *ad hoc* ones). As mathematical logic formalised mathematical reasoning, logic for computer science should deal with natural languages of computer science. Of course, computer science is young, and we should not expect the same kind of maturity that the language of mathematicians had reached when logic began. That said, I will consider CCS a natural language to close as much as possible on. As we will see, one of the main problems we have to deal with is the difference between the logical notion of sequentiality of seq, and the operational one of CCS's prefix combinator.

Point 2: In the calculus of structures, even more than in the sequent calculus, the bottom-up construction of proofs is a very non-deterministic process; this is due to the fact that inference rules may be applied anywhere deep in a structure. If this non-determinism is not tamed, our ability to design concurrent algorithms is severely hampered. Here I will solve part of this problem: to establish the operational correspondence between seq and prefix we have to coerce the search for proofs, otherwise the order induced by seq is not respected by the computational interpretation of proofs. This aspect is solved *logically*: I will show a system, called BVL, which is equivalent to SBV but which generates only those proofs that correspond to computations respecting the time-order induced by the prefixing. I will show the correspondence to CCS of this intermediate system.

Still, BVL generates more proofs than desirable for just an operational account, and the best answer to this problem should come by further applying methods inspired by Miller's uniform proofs. We will not deal with this in the present

paper, although I argue that this operation is entirely feasible because: 1) the calculus of structures is *more general* than the sequent calculus, so the methods for the sequent calculus should work as well; 2) our system is an extension of multiplicative linear logic, which so far has been the most successful logical system *vis-à-vis* the uniform proofs [12].

Point 3: Recent work by Guglielmi and Straßburger provides the extension: they designed a Turing-equivalent system, called SNEL, which conservatively extends SBV with exponentials [9]. Since we find there the usual exponential of linear logic, it should be possible to map fixpoint operators by simple, known replication techniques. SNEL is also a conservative extension of MELL, the multiplicative-exponential fragment of linear logic, amenable to the uniform proof reduction mentioned above. CCS choice operator requires additives: a presentation of full linear logic is provided in [15]; then we can borrow techniques from [11].

For those reasons, this paper establishes the first of what I believe is a two-step move towards the first abstract logic programming system directly corresponding to CCS and similar process algebras. More in detail, these are the contributions of this paper:

1 A logical system in the calculus of structures, BVL, which is equivalent to SBV and which shows a general technique for limiting non-determinism in the case of a non-commutative self-dual logical operator. This is a purely proof-theoretical result (Section 3).

2 A simple process algebra, $\mathsf{PA}_{\mathsf{BV}}$, corresponding to CCS restricted to the sequential and parallel operators, which is exactly captured by BVL: 1) Every terminating computation in it corresponds to a proof of BVL. 2) For every (legal) expression provable in BVL there is a corresponding terminating computation (Section 4).

Compared to some previous work, notably by Miller [11] and Guglielmi [5, 6], my approach has a distinctive, important feature: sequentiality is not obtained through axioms, or through an encoding, rather it is realised by a logical operator in the system. Despite the simplicity of the system, getting cut elimination has proved extremely difficult (it turned out to be impossible in the sequent calculus) and required the development of the calculus of structures.

This effort gives us an important property in exchange. As I will argue later in the paper, we will be able to manipulate proofs at various levels of abstraction: 1) There is the concrete level of BVL, where a proof closely corresponds to a computation. 2) More abstractly, we can use a restriction of SBV called BV, where we are free to exchange messages disregarding the actual ordering of the computation; here, for example, we could verify what happens towards the end of a computation without being forced to execute its beginning. 3) Even more abstractly, we could use in addition a new admissible rule which allows us to separate certain threads of a computation when performing an analysis. 4) Finally, we can use cut rules (in various forms), so reducing dramatically the search space.

As is typical in the calculus of structures, there is in fact a whole hierarchy of equivalent systems, generated as a consequence of the more general kind of cut elimination we have in this formalism. The smallest system is the concrete one,

corresponding to computations; all the others can be used for analysis, verification, and the like.

2 Basic Definitions

In this section I will shortly present definitions and results that the reader can find in more extensive details in [7] and [8]. I call *calculus* a formalism, like natural deduction or the sequent calculus, for specifying logical systems. A *system* in the calculus of structures is defined by a language of structures, an equational theory over structures, and a collection of inference rules. The equational theory serves just the purpose of handling simple decidable properties, like commutativity or idempotency of logical operators, something that in the sequent calculus is usually implicitly assumed. It also defines negation, as is typical in linear logic.

Let us first define the language of structures of BV. Intuitively, $[S_1, \ldots, S_h]$ corresponds to a sequent $\vdash S_1, \ldots, S_h$ or, equivalently to the formula $S_1 \parr \cdots \parr S_h$. The structure $(S_1, \ldots, S_h)$ corresponds to $S_1 \otimes \cdots \otimes S_h$. The structure $\langle S_1; \ldots; S_h \rangle$ has no correspondence in linear logic, it should be considered the *sequential* or *non-commutative* composition of $S_1, \ldots, S_h$.

2.1 Definition We consider a set $\mathcal{A}$ of countably many *positive atoms* and *negative atoms*, denoted by a, b, c, ... *Structures* are denoted with S, P, Q, R, T, U and V. The structures of the *language* BV are generated by

$$S ::= \circ \mid a \mid \langle \underbrace{S; \ldots; S}_{>0} \rangle \mid [\underbrace{S, \ldots, S}_{>0}] \mid (\underbrace{S, \ldots, S}_{>0}) \mid \bar{S} \quad ,$$

where $\circ$, the *unit*, is not an atom; $\langle S_1; \ldots; S_h \rangle$ is a *seq structure*, $[S_1, \ldots, S_h]$ is a *par structure* and $(S_1, \ldots, S_h)$ is a *copar structure*; $\bar{S}$ is the *negation* of the structure S. The notation $S\{\ \}$ stands for a structure with a hole that is not in the scope of a negation, and denotes the *context* of the structure R in $S\{R\}$; we also say that the structure R is a *substructure* of $S\{R\}$. We drop contextual parentheses whenever structural parentheses fill exactly the hole: for instance $S[R, T]$ stands for $S\{[R, T]\}$.

Inference rules assume a peculiar shape in our formalism: they typically have the form $\rho \dfrac{S\{T\}}{S\{R\}}$, which stands for a scheme ρ, stating that if a structure matches R, in a context $S\{\ \}$, then it can be replaced by T without acting in the context at all (and analogously if one prefers a top-down reading). A rule is a way to implement any axiom $T \Rightarrow R$, where $\Rightarrow$ stands for the implication we model in the system, but it would be simplistic to regard a rule as a different notation for axioms. The entire design process of rules is done for having cut elimination and the subformula property; these proof theoretical properties are foundational for proof search and abstract logic programming. A *derivation* is a composition of instances of inference rules and a *proof* is a derivation free from hypotheses: the shape of rules confers to derivations (but not to proofs) a vertical symmetry.

2.2 Definition An (*inference*) *rule* is any scheme $\rho\dfrac{T}{R}$, where ρ is the *name* of the rule, T is its *premise* and R is its *conclusion*; at most one between R and T may be missing. A set of rules defines a (*formal*) *system*, denoted by $\mathscr{S}$. A *derivation* in a system $\mathscr{S}$ is a finite chain of instances of rules of $\mathscr{S}$, is denoted by Δ and can consist of just one structure. Its topmost and bottommost structures are respectively called *premise* and *conclusion*. A derivation Δ in $\mathscr{S}$ whose premise is T and conclusion is R is denoted by $\Delta\dfrac{\|T}{R}\!\!\|_{\mathscr{S}}$.

It is customary in the calculus of structures first to define symmetric systems, returning just derivations, and only afterwards to break the symmetry by adding an (asymmetric) axiom. Symmetric systems are obtained by considering for each rule also its *corule*, defined by swapping and negating premise and conclusion. Hence, we typically deal with *pairs* of rules, $\rho{\downarrow}\,\dfrac{S\{T\}}{S\{R\}}$ (down version) and $\rho{\uparrow}\,\dfrac{S\{\bar{R}\}}{S\{\bar{T}\}}$ (up version), that make the system closed by contraposition. When the up and down versions coincide, the rules are self-dual, and in this case we will omit the arrows.

We now define system BV by extracting it from its symmetric version SBV. In SBV we distinguish a fragment, called *interaction*, which deals solely with negation; the rest of the system, the *structure* fragment, deals with logical relations. In analogy with sequent calculus presentations, the interaction fragment corresponds to the rules dealing with identity and cut, and the structure fragment to logical (and structural) rules. Note that in the calculus of structures rules are defined on complex contexts: *pairs* of logical relations are taken simultaneously into account.

2.3 Definition The structures of the language BV are equivalent modulo the relation $=$, defined at the left of Fig. 1. By $\vec{R}$, $\vec{T}$ and $\vec{U}$ we denote finite, non-empty sequences of structures (sequences may contain ',' or ';' separators as appropriate in the context). Structures whose only negated substructures are atoms are said to be *in normal form*. At the right of the figure *system* SBV is shown (*symmetric basic system* V). The rules $\mathsf{ai}{\downarrow}$, $\mathsf{ai}{\uparrow}$, s, $\mathsf{q}{\downarrow}$ and $\mathsf{q}{\uparrow}$ are called respectively *atomic interaction*, *atomic cut*, *switch*, *seq* and *coseq*. The *down fragment* of SBV is $\{\mathsf{ai}{\downarrow},\mathsf{s},\mathsf{q}{\downarrow}\}$, the *up fragment* is $\{\mathsf{ai}{\uparrow},\mathsf{s},\mathsf{q}{\uparrow}\}$.

It helps intuition always to consider structures in normal form, where not otherwise indicated. There is a straightforward two-way correspondence between structures not involving seq and formulae of multiplicative linear logic (MLL) in the version including mix and nullary mix: for example $[(a,\bar{b}),c,\bar{d}]$ corresponds to $((a\otimes b^{\perp})\parr c\parr d^{\perp})$, and vice versa. Units are mapped into $\circ$, since $1\equiv\perp$, when mix and nullary mix are present [1]. The reader can check that the equations in Fig. 1 correspond to equivalences in MLL plus mix and nullary mix, disregarding seq, and that rules correspond to valid implications.

Our three logical relations share a common self-dual unit $\circ$, which can be regarded as the empty sequence; it gives us flexibility in the application of rules.

Associativity

$$\langle \vec{R}; \langle \vec{T}; \vec{U} \rangle \rangle = \langle \vec{R}; \vec{T}; \vec{U} \rangle$$
$$[\vec{R}, [\vec{T}]] = [\vec{R}, \vec{T}]$$
$$(\vec{R}, (\vec{T})) = (\vec{R}, \vec{T})$$

Commutativity

$$[\vec{R}, \vec{T}] = [\vec{T}, \vec{R}]$$
$$(\vec{R}, \vec{T}) = (\vec{T}, \vec{R})$$

Unit

$$\langle \circ; \vec{R} \rangle = \langle \vec{R}; \circ \rangle = \langle \vec{R} \rangle$$
$$[\circ, \vec{R}] = [\vec{R}]$$
$$(\circ, \vec{R}) = (\vec{R})$$

Negation

$$\bar{\circ} = \circ$$
$$\overline{\langle R; T \rangle} = \langle \bar{R}; \bar{T} \rangle$$
$$\overline{[R, T]} = (\bar{R}, \bar{T})$$
$$\overline{(R, T)} = [\bar{R}, \bar{T}]$$
$$\bar{\bar{R}} = R$$

Singleton

$$\langle R \rangle = [R] = (R) = R$$

Contextual Closure

if $R = T$
then $S\{R\} = S\{T\}$

$$\mathsf{ai}{\downarrow}\,\frac{S\{\circ\}}{S[a, \bar{a}]} \qquad \mathsf{ai}{\uparrow}\,\frac{S(a, \bar{a})}{S\{\circ\}}$$

Interaction

Structure

$$\mathsf{s}\,\frac{S([R, U], T)}{S[(R, T), U]}$$

$$\mathsf{q}{\downarrow}\,\frac{S\langle [R, U]; [T, V] \rangle}{S[\langle R; T \rangle, \langle U; V \rangle]} \qquad \mathsf{q}{\uparrow}\,\frac{S(\langle R; U \rangle, \langle T; V \rangle)}{S\langle (R, T); (U, V) \rangle}$$

Fig. 1 **Left:** *Syntactic equivalence* $=$ *for* BV **Right:** *System* SBV

For example, consider the following derivation:

$$\begin{array}{l} \mathsf{q}{\uparrow}\,\dfrac{(a, b)}{\langle a; b \rangle} \\ \mathsf{q}{\downarrow}\,\dfrac{\langle a; b \rangle}{[a, b]} \end{array} \quad = \quad \mathsf{q}{\uparrow}\,\frac{(\langle a; \circ \rangle, \langle \circ; b \rangle)}{\langle [a, \circ]; [\circ, b] \rangle = \langle (a, \circ); (\circ, b) \rangle} \quad \mathsf{q}{\downarrow}\,\frac{}{[\langle a; \circ \rangle, \langle \circ; b \rangle]} \quad .$$

Looking at the rules of system SBV, we note that all of them, apart from the cut rule, guarantee the subformula property: the premise only involves substructures of the structures of the conclusion.

The rules $\mathsf{i}{\downarrow}\,\dfrac{S\{\circ\}}{S[R, \bar{R}]}$ and $\mathsf{i}{\uparrow}\,\dfrac{S(R, \bar{R})}{S\{\circ\}}$ define respectively general forms of *interaction* and *cut*: as shown in [7, 8], they are admissible, respectively, for the down and up fragment of SBV.

So far we have dealt with SBV, a top-down symmetric system, lacking any notion of proof. Particularly relevant for provability is a study of permutability and admissibility of rules: the symmetric system is simplified into an equivalent minimal one, by discarding the entire fragment of up rules. Behind this, is that $T \Rightarrow R$ and $\bar{R} \Rightarrow \bar{T}$ are equivalent statements in many logics. Related to this phenomenon, systems in the calculus of structures have two distinctive features:

1 The cut rule splits into several up rules, and since we can eliminate up rules successively and independently one from the other, the cut elimination argument becomes modular. In our case $\mathsf{i}{\uparrow}$ can be decomposed into $\mathsf{ai}{\uparrow}, \mathsf{s}$ and $\mathsf{q}{\uparrow}$, in every derivation.

2 Adding up rules to the minimal system, while preserving provability, allows to define a broader range of equivalent systems than what we might expect in more traditional calculi, like sequent calculus (or natural deduction).

2.4 Definition The following (logical axiom) rule is called *unit*: $\circ{\downarrow}\,\dfrac{}{\circ}$. The system in Fig. 2 is called *system* BV (*basic system* V).

Note that system BV is cut-free, and every rule has the subformula property.

$$\circ\!\downarrow \frac{}{\circ} \qquad \mathsf{ai}\!\downarrow \frac{S\{\circ\}}{S[a,\bar{a}]} \qquad \mathsf{s}\,\frac{S([R,U],T)}{S[(R,T),U]} \qquad \mathsf{q}\!\downarrow \frac{S\langle[R,U];[T,V]\rangle}{S[\langle R;T\rangle,\langle U;V\rangle]}$$

Fig. 2 *System* BV

2.5 Definition A *proof* is a derivation whose topmost inference rule is an instance of the unit rule. Proofs are denoted with Π. A formal system $\mathscr{S}$ *proves* R if there is in $\mathscr{S}$ a proof Π whose conclusion is R, written $\begin{array}{c}\Pi\\ \|\\ S\end{array}\!\!\mathscr{S}$. Two systems are *equivalent* if they prove the same structures.

Observe that $\circ\!\downarrow$ can only occur once in a derivation, and only at the top.

This is the cut elimination theorem, in a much more general form than possible in the sequent calculus:

2.6 Theorem *All the following systems are equivalent*: BV, $\mathsf{BV}\cup\{\mathsf{q}\uparrow\}$, $\mathsf{BV}\cup\{\mathsf{ai}\uparrow\}$, $\mathsf{BV}\cup\{\mathsf{i}\uparrow\}$, *and* $\mathsf{SBV}\cup\{\circ\!\downarrow\}$.

In addition, and according to the correspondence mentioned above, we have that BV is a conservative extension of MLL plus mix and nullary mix.

3 Restricting Interaction

In this section we will see a system equivalent to BV, and so to all systems equivalent to it, in which interaction is limited to certain contexts only. This limitation will be instrumental in showing the correspondence to CCS. Intuitively, in CCS interaction happens in the order induced by prefixing; by restricting interaction in BV, we force this ordering. Some proofs in the following are very sketchy, due to length constraints. I tried to put the emphasis on the techniques that are closer to our process algebra.

3.1 Definition The structure context $S\{\ \}$ is a *right context* if there are no structure $R \not= \circ$ and no contexts $S'\{\ \}$ and $S''\{\ \}$ such that $S\{\ \} = S'\langle R; S''\{\ \}\rangle$. Right contexts are also denoted by $S\{\ \}^{\llcorner}$, where the L stands for (hole at the) left. We tag with $^{\llcorner}$ structural parentheses instead of contextual ones whenever possible: for example $S[R,T]^{\llcorner}$ stands for $S\{[R,T]\}^{\llcorner}$.

For example $S_1\{\ \}^{\llcorner} = [a,b,\langle\{\ \};c\rangle]$, $S_2\{\ \}^{\llcorner} = (a,\{\ \},b)$ and $S_3\{\ \}^{\llcorner} = \langle[a,\{\ \}];b\rangle$ are right contexts, whilst $[a,(b,\langle c;\{\ \}\rangle)]$ and $\langle(a,[b,c]);\{\ \}\rangle$ are not.

3.2 Definition The next rule is called *left atomic interaction*: $\mathsf{ai}\!\downarrow_{\llcorner} \dfrac{S\{\circ\}^{\llcorner}}{S[a,\bar{a}]^{\llcorner}}$; $[a,\bar{a}]$ is its *redex*. The system $\{\circ\!\downarrow, \mathsf{ai}\!\downarrow_{\llcorner}, \mathsf{q}\!\downarrow, \mathsf{s}\}$ is called *system* BVL.

Trivially, instances of $\mathsf{ai}\!\downarrow_{\llcorner}$ are instances of $\mathsf{ai}\!\downarrow$, and hence any proof in BVL is also a proof in BV.

We introduce some terminology for our coming analysis of permutability.

3.3 Definition A rule ρ *permutes by* $\mathscr{S}$ *over* ρ' if for all $\rho\,\dfrac{\rho'\,\dfrac{Q}{U}}{P}$ there is $\rho\,\dfrac{Q}{V}$ $\begin{array}{c}\|\\ P\end{array}\!\!\mathscr{S}\cup\{\rho'\}$, for some V.

3.4 Lemma *The rule* $\mathsf{ai}{\downarrow}$ *permutes by* $\{\mathsf{q}{\downarrow}\}$ *over* $\mathsf{ai}{\downarrow_{\mathsf{L}}}$.

Proof Consider $\Delta = \mathsf{ai}{\downarrow}\dfrac{\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{Q}{S\{\circ\}}}{S[a,\bar{a}]}$. We reason about the position of the redex of $\mathsf{ai}{\downarrow_{\mathsf{L}}}$ in $S\{\circ\}$. The following cases exhaust all possibilities:

1 The redex of $\mathsf{ai}{\downarrow_{\mathsf{L}}}$ is inside context $S\{\ \}$:

$$\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{\mathsf{ai}{\downarrow}\dfrac{S'\{\circ\}}{S\{\circ\}}}{S[a,\bar{a}]} \quad \text{yields} \quad \mathsf{ai}{\downarrow}\dfrac{\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{S'\{\circ\}}{S'[a,\bar{a}]}}{S[a,\bar{a}]} \quad .$$

2 Otherwise, there are only three possibilities:

1 $S\{\ \} = S'[b, \langle\{\ \}; \bar{b}\rangle]$, for some b; in this case

$$\Delta = \mathsf{ai}{\downarrow}\dfrac{\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{S'\{\circ\}^{\mathsf{L}}}{S'[b,\bar{b}]^{\mathsf{L}}}}{S'[b,\langle[a,\bar{a}];\bar{b}\rangle]^{\mathsf{L}}} \quad \text{trivially yields} \quad \mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{S'\{\circ\}^{\mathsf{L}}}{S'[b,\bar{b}]^{\mathsf{L}}}}{S'[b,\langle[a,\bar{a}];\bar{b}\rangle]^{\mathsf{L}}} \quad .$$

2 $S\{\ \} = S'[b, \langle\bar{b}; \{\ \}\rangle]$, for some b; in this case

$$\Delta = \mathsf{ai}{\downarrow}\dfrac{\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{S'\{\circ\}^{\mathsf{L}}}{S'[b,\bar{b}]^{\mathsf{L}}}}{S'[b,\langle\bar{b};[a,\bar{a}]\rangle]^{\mathsf{L}}} \quad \text{yields} \quad \dfrac{\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{S'\{\circ\}^{\mathsf{L}}}{S'[a,\bar{a}]^{\mathsf{L}}}}{\mathsf{q}{\downarrow}\dfrac{S'\langle[b,\bar{b}];[a,\bar{a}]\rangle^{\mathsf{L}}}{S'[b,\langle\bar{b};[a,\bar{a}]\rangle]^{\mathsf{L}}}} \quad .$$

3 $S\{\ \} = S'[b, (\{\ \}, \bar{b})]$, for some b; in this case

$$\Delta = \mathsf{ai}{\downarrow}\dfrac{\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{S'\{\circ\}^{\mathsf{L}}}{S'[b,\bar{b}]^{\mathsf{L}}}}{S'[b,([a,\bar{a}],\bar{b})]^{\mathsf{L}}} \quad \text{trivially yields} \quad \mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{\mathsf{ai}{\downarrow_{\mathsf{L}}}\dfrac{S'\{\circ\}^{\mathsf{L}}}{S'[b,\bar{b}]^{\mathsf{L}}}}{S'[b,([a,\bar{a}],\bar{b})]^{\mathsf{L}}} \quad .$$

□

3.5 Lemma *The rule* $\mathsf{ai}{\downarrow}$ *permutes by* $\{\mathsf{q}{\uparrow},\mathsf{s}\}$ *over the rules* $\mathsf{q}{\downarrow}$, $\mathsf{q}{\uparrow}$ *and* s.

Proof We first prove that for every $S\{\ \}$ and R there exists a derivation $\dfrac{(S\{\circ\},R)}{S\{R\}}\Big\|_{\{\mathsf{q}{\uparrow},\mathsf{s}\}}$ (easy structural induction on $S\{\ \}$); then for every $\rho \in \{\mathsf{q}{\downarrow},\mathsf{q}{\uparrow},\mathsf{s}\}$ we have:

$$\rho\dfrac{\mathsf{ai}{\downarrow}\dfrac{Q}{S\{\circ\}}}{S[a,\bar{a}]} \quad \text{yields} \quad \begin{array}{c}\mathsf{ai}{\downarrow}\dfrac{Q}{(Q,[a,\bar{a}])}\\ \rho\dfrac{}{(S\{\circ\},[a,\bar{a}])}\\ \Big\|_{\{\mathsf{q}{\uparrow},\mathsf{s}\}}\\ S[a,\bar{a}]\end{array} \quad .$$

□

Then, trivially, from Lemmas 3.4 and 3.5:

3.6 Theorem *The rule* $\mathsf{ai}{\downarrow}$ *permutes by* $\{\mathsf{q}{\downarrow},\mathsf{q}{\uparrow},\mathsf{s}\}$ *over* $\mathsf{ai}{\downarrow_{\mathsf{L}}},\mathsf{q}{\downarrow},\mathsf{q}{\uparrow}$ *and* s.

3.7 Theorem *If there is a proof for R in* BV*, then there is a proof for R in* $\mathsf{BVL} \cup \{\mathsf{q}\uparrow\}$.

Proof The topmost instance of $\mathsf{ai}{\downarrow}$ in a proof is also an instance of $\mathsf{ai}{\downarrow_{\mathrm{L}}}$. Transform the given proof as follows: Take the topmost instance of an $\mathsf{ai}{\downarrow}$ rule which is not already an $\mathsf{ai}{\downarrow_{\mathrm{L}}}$ instance and permute it up, by Theorem 3.6, until it becomes an instance of $\mathsf{ai}{\downarrow_{\mathrm{L}}}$ (which always happens when the instance reaches the top of a proof). Proceed inductively. □

For example, the proof on the left, where we have already renamed the topmost instance of $\mathsf{ai}{\downarrow}$ as $\mathsf{ai}{\downarrow_{\mathrm{L}}}$, is successively transformed as follows:

$$\mathsf{ai}{\downarrow}\dfrac{\dfrac{\dfrac{\circ\downarrow\dfrac{}{\circ}}{\mathsf{ai}{\downarrow_{\mathrm{L}}}\dfrac{}{[c,\bar{c}]}}}{\mathsf{ai}{\downarrow}\dfrac{}{[c,\langle\bar{c};[b,\bar{b}]\rangle]}}}{[c,\langle\bar{c};[b,(\bar{b},[a,\bar{a}])]\rangle]} \quad\rightarrow\quad \mathsf{ai}{\downarrow}\dfrac{\dfrac{\circ\downarrow\dfrac{}{\circ}}{\mathsf{ai}{\downarrow_{\mathrm{L}}}\dfrac{}{[b,\bar{b}]}}}{\mathsf{q}{\downarrow}\dfrac{\mathsf{ai}{\downarrow_{\mathrm{L}}}\dfrac{}{\langle[c,\bar{c}];[b,\bar{b}]\rangle}}{[c,\langle\bar{c};[b,\bar{b}]\rangle]}}}{[c,\langle\bar{c};[b,(\bar{b},[a,\bar{a}])]\rangle]} \quad\rightarrow$$

$$\mathsf{ai}{\downarrow}\dfrac{\mathsf{q}{\downarrow}\dfrac{\mathsf{ai}{\downarrow}\dfrac{\mathsf{ai}{\downarrow_{\mathrm{L}}}\dfrac{\circ\downarrow\dfrac{}{\circ}}{[b,\bar{b}]}}{\langle[c,\bar{c}];[b,\bar{b}]\rangle}}{\langle[c,\bar{c}];[b,(\bar{b},[a,\bar{a}])]\rangle}}{[c,\langle\bar{c};[b,(\bar{b},[a,\bar{a}])]\rangle]} \quad\rightarrow\quad \mathsf{ai}{\downarrow}\dfrac{\mathsf{q}{\downarrow}\dfrac{\mathsf{ai}{\downarrow_{\mathrm{L}}}\dfrac{\mathsf{ai}{\downarrow_{\mathrm{L}}}\dfrac{\circ\downarrow\dfrac{}{\circ}}{[b,\bar{b}]}}{[b,(\bar{b},[a,\bar{a}])]}}{\langle[c,\bar{c}];[b,(\bar{b},[a,\bar{a}])]\rangle}}{[c,\langle\bar{c};[b,(\bar{b},[a,\bar{a}])]\rangle]} \quad .$$

We need to refine the preceding theorem such that we can get rid of the $\mathsf{q}\uparrow$ rule in our system.

3.8 Theorem *If there is a proof for R in* BV*, and no copar structure appears in R, then there is a proof for R in* BVL.

Proof Take the given proof for R and transform it into one in $\mathsf{BVL} \cup \{\mathsf{q}\uparrow\}$, by Theorem 3.7. Since no copar appears in R, the bottommost instance of $\mathsf{q}\uparrow$ in the proof must necessarily be as in

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{}\,{\scriptstyle \mathsf{BVL}\cup\{\mathsf{q}\uparrow\}}}{S(T,U)}}{\mathsf{q}\uparrow\dfrac{}{S\langle T;U\rangle}}}{}\,{\scriptstyle \mathsf{BVL}}}{}}{R} \quad .$$

Transform the proof by upwardly changing (T,U) into $\langle T;U\rangle$, and correspondingly transforming s instances into $\mathsf{q}{\downarrow}$ instances. This eliminates one instance of $\mathsf{q}\uparrow$. Possibly, some instances of $\mathsf{ai}{\downarrow_{\mathrm{L}}}$ become simple $\mathsf{ai}{\downarrow}$. Rearrange them until all are again $\mathsf{ai}{\downarrow_{\mathrm{L}}}$ and repeat the procedure until all $\mathsf{q}\uparrow$ instances are eliminated. □

At this time I don't know whether it is possible to lift the restriction on R containing no copars. I believe that it is possible, but the proof does not look easy.

Laws for expressions

$$E \mid \circ = E$$
$$E \mid E' = E' \mid E$$
$$E \mid (E' \mid E'') = (E \mid E') \mid E''$$

Law for action sequences

$$\alpha_1; \ldots; \alpha_{i-1}; \circ; \alpha_i; \ldots; \alpha_n = \alpha_1; \ldots; \alpha_n$$

$$\mathsf{Cp}\ \frac{}{a.E \mid F \xrightarrow{a} E \mid F} \qquad \mathsf{Cs}\ \frac{E \xrightarrow{a} E' \quad F \xrightarrow{\bar{a}} F'}{E \mid F \xrightarrow{\circ} E' \mid F'}$$

Fig. 3 **Left:** *Syntactic equivalences for* $\mathsf{PA_{BV}}$ **Right:** *Transition rules for* $\mathsf{PA_{BV}}$

4 Relations with a Simple Process Algebra

4.1 Completeness

We now introduce some definitions and notation for a simple process algebra $\mathsf{PA_{BV}}$ corresponding to the CCS fragment of prefixing and parallel composition.

4.1.1 Definition Let $\mathcal{L} = (\mathcal{A}/=) \cup \{\circ\}$ be the set of *labels* or *actions*, where $\circ$ is called the *internal* (or *silent*) action; we denote actions by α. The *process expressions* of $\mathsf{PA_{BV}}$, denoted by E and F, are generated by

$$E ::= \circ \mid a.E \mid (E \mid E) \quad ,$$

where the combinators '.' and '|' are called respectively *prefix* and *composition*, and prefix is stronger than composition. We will consider expressions equivalent up to the laws defined at the left in Fig. 3. We denote the set of expressions by $\mathcal{E}_{\mathsf{PA}}$. At the right of Fig. 3 the transition rules of $\mathsf{PA_{BV}}$ are defined: Cp is called *prefix* and Cs is called *synchronisation*.

Operational semantics is given by way of the labelled transition system $(\mathcal{E}_{\mathsf{PA}}, \mathcal{L}, \{\xrightarrow{\alpha} : \alpha \in \mathcal{L}\})$. We introduce some basic terminology and notation.

4.1.2 Definition In the *computation* $E \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} F$ we call $\alpha_1; \ldots; \alpha_n$ an *action sequence* of E; action sequences are considered equivalent up to the law at the left in Fig. 3; action sequences are denoted by $\vec{s}$; if $n = 0$ then E is the empty computation, its action sequence is empty and is denoted by ϵ. *Terminating* computations are those whose last expression is $\circ$. A computation $E \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} F$ can also be written $E \xrightarrow{\alpha_1;\ldots;\alpha_n} F$.

The reader will have no trouble in verifying that our process algebra indeed is equivalent to the fragment of CCS with prefix and parallel composition, as is presented, for example, in [14]. We make no distinction between 0 and τ, they both are collapsed into the unit $\circ$.

4.1.3 Definition The function $\underline{\cdot}_{\mathsf{s}}$ maps the expressions in $\mathcal{E}_{\mathsf{PA}}/=$ and the action sequences in $\mathcal{L}^*/=$ into structures of BV according to the following inductive definition:

$$\underline{\circ}_{\mathsf{s}} = \circ \quad , \qquad \underline{\epsilon}_{\mathsf{s}} = \circ \quad ,$$
$$\underline{a.E}_{\mathsf{s}} = \langle a; \underline{E}_{\mathsf{s}} \rangle \quad , \qquad \underline{a}_{\mathsf{s}} = a \quad ,$$
$$\underline{E \mid F}_{\mathsf{s}} = [\underline{E}_{\mathsf{s}}, \underline{F}_{\mathsf{s}}] \quad ; \qquad \underline{\alpha_1; \ldots; \alpha_n}_{\mathsf{s}} = \langle \underline{\alpha_1}_{\mathsf{s}}; \ldots; \underline{\alpha_n}_{\mathsf{s}} \rangle \quad .$$

4.1.4 Theorem *For every computation* $E_0 \xrightarrow{\vec{s}} E_n$ *there is a derivation* $\dfrac{\underline{E_n}_{\mathsf{S}}}{\big\|_{\mathsf{BVL}}}$. $[\underline{E_0}_{\mathsf{S}}, \underline{\vec{s}}_{\mathsf{S}}]$

Proof By induction on n. If $n = 0$ take the derivation $\underline{E_0}_{\mathsf{S}}$. The inductive cases are:

1 $E_0 \xrightarrow{a} E_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} E_n$: It must be $E_0 = a.E \mid F$, for some E and F, and $E_1 = E \mid F$. Let $S = \underline{\overline{\alpha_2; \ldots; \alpha_n}}_{\mathsf{S}}$; we can build:

$$\dfrac{\dfrac{\dfrac{\underline{E_n}_{\mathsf{S}}}{\big\|_{\mathsf{BVL}}}}{[\underline{E}_{\mathsf{S}}, \underline{F}_{\mathsf{S}}, S]}}{[\langle [a, \bar{a}]; [\underline{E}_{\mathsf{S}}, S]\rangle, \underline{F}_{\mathsf{S}}]}\,\mathsf{ai}{\downarrow}_{\mathsf{L}}}{[\langle a; \underline{E}_{\mathsf{S}}\rangle, \underline{F}_{\mathsf{S}}, \langle \bar{a}; S\rangle]}\,\mathsf{q}{\downarrow} \quad .$$

2 $E_0 \xrightarrow{\circ} E_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} E_n$: It must be $E_0 = E \mid F$, $E_1 = E' \mid F'$, $E = a.E'' \mid F''$, $E' = E'' \mid F''$, $F = \bar{a}.E''' \mid F'''$ and $F' = E''' \mid F'''$. Let $S = \underline{\overline{\alpha_2; \ldots; \alpha_n}}_{\mathsf{S}}$; we can build:

$$\dfrac{\dfrac{\dfrac{\underline{E_n}_{\mathsf{S}}}{\big\|_{\mathsf{BVL}}}}{[\underline{E''}_{\mathsf{S}}, \underline{F''}_{\mathsf{S}}, \underline{E'''}_{\mathsf{S}}, \underline{F'''}_{\mathsf{S}}, S]}}{[\langle [a, \bar{a}]; [\underline{E''}_{\mathsf{S}}, \underline{E'''}_{\mathsf{S}}]\rangle, \underline{F''}_{\mathsf{S}}, \underline{F'''}_{\mathsf{S}}, S]}\,\mathsf{ai}{\downarrow}_{\mathsf{L}}}{[\langle a; \underline{E''}_{\mathsf{S}}\rangle, \underline{F''}_{\mathsf{S}}, \langle \bar{a}; \underline{E'''}_{\mathsf{S}}\rangle, \underline{F'''}_{\mathsf{S}}, S]}\,\mathsf{q}{\downarrow} \quad .$$

□

4.1.5 Corollary *For every terminating computation in* $\mathsf{PA}_{\mathsf{BV}}$ *there exists a proof in* BVL.

4.2 Soundness

Now comes the tricky part. We want to map provable structures of BV to terminating computations of $\mathsf{PA}_{\mathsf{BV}}$ and, of course, we need a linguistic restriction on BV, which be determined by the grammar for expressions and action sequences of $\mathsf{PA}_{\mathsf{BV}}$. This restriction provides the legal set of structures we may use.

4.2.1 Definition The set $\mathcal{E}_{\mathsf{BV}}$ of *process structures* is the set of structures obtained by

$$P ::= \circ \mid \langle a; P\rangle \mid [P, P] \quad .$$

The function $\underline{\cdot}_{\mathsf{E}}$ maps the structures in $\mathcal{E}_{\mathsf{BV}}/{=}$ into expressions in $\mathcal{E}_{\mathsf{PA}}/{=}$ as follows:

$$\underline{\circ}_{\mathsf{E}} = \circ \quad ,$$

$$\underline{\langle a; P\rangle}_{\mathsf{E}} = a.\underline{P}_{\mathsf{E}} \quad ,$$

$$\underline{[P, Q]}_{\mathsf{E}} = \underline{P}_{\mathsf{E}} \mid \underline{Q}_{\mathsf{E}} \quad .$$

4.2.2 Theorem *Given the process structure P and the proof* $\dfrac{\top\!\!\top_{\mathsf{BVL}}}{[P, \langle a_1; \ldots; a_n\rangle]}$, *for* $n \geq 0$, *there exists a computation* $P_0 \xrightarrow{\vec{s}} \circ$, *where* $P_0 = \underline{P}_{\mathsf{E}}$ *and* $\underline{\vec{s}}_{\mathsf{S}} = \overline{\langle a_1; \ldots; a_n\rangle}$.

Proof By induction on the size of P. If $P = \circ$ then P_0 is the computation. Otherwise, consider the given proof, where the bottommost instance of $\mathsf{ai}{\downarrow_{\mathsf{L}}}$ has been singled out:

$$\dfrac{\dfrac{\dfrac{}{S\{\circ\}}\top\!\!\top_{\mathsf{BVL}}}{S[b,\bar{b}]^{\mathsf{L}}}\,\mathsf{ai}{\downarrow_{\mathsf{L}}}}{[P, \langle a_1; \ldots; a_n\rangle]}\,\Delta\left\|_{\mathsf{BVL}\setminus\{\mathsf{ai}{\downarrow_{\mathsf{L}}}\}}\right.$$

Let us mark into Δ all occurrences of b and $\bar{b}$, as in $b^\bullet$ and $\bar{b}^\bullet$. Only two possibilities might occur:

1 One marked atom occurs in P an another occurs in $\langle a_1; \ldots; a_n\rangle$: In this case it must be $P = [\langle b^\bullet; P'\rangle, P'']$, for some P' and P'', and $a_1 = \bar{b}^\bullet$. Any other possibility would result in violating the condition of $S\{\ \}^{\mathsf{L}}$ being a right context (to see this, check carefully the rules of $\mathsf{BVL} \setminus \{\mathsf{ai}{\downarrow_{\mathsf{L}}}\}$ and see how they always respect seq orderings). Then replace all marked atoms by $\circ$, and remove all trivial occurrences of rule instances that result from this, including the $\mathsf{ai}{\downarrow_{\mathsf{L}}}$ instance. We still have a proof and $[P', P'']$ is a process structure, so we can apply the induction hypothesis on the proof

$$\dfrac{}{[P', P'', \langle a_2; \ldots; a_n\rangle]}\top\!\!\top_{\mathsf{BVL}}\ .$$

We get $b.\underline{P'}_{\mathsf{E}} \mid \underline{P''}_{\mathsf{E}} \xrightarrow{b} \underline{P'}_{\mathsf{E}} \mid \underline{P''}_{\mathsf{E}} \xrightarrow{\vec{s}'} \circ$, where $\underline{\vec{s}'}_{\mathsf{S}} = \overline{\langle a_2; \ldots; a_n\rangle}$.

2 Both marked atoms occur in P: It must be $P = [\langle b^\bullet; P'\rangle, \langle \bar{b}^\bullet; P''\rangle, P''']$, for the same reasons as above. By substituting $b^\bullet$ and $\bar{b}^\bullet$ by $\circ$, analogously as above, we can get, by induction hypothesis, the computation $b^\bullet.\underline{P'}_{\mathsf{E}} \mid \bar{b}^\bullet.\underline{P''}_{\mathsf{E}} \mid \underline{P'''}_{\mathsf{E}} \xrightarrow{\circ} \underline{P'}_{\mathsf{E}} \mid \underline{P''}_{\mathsf{E}} \mid \underline{P'''}_{\mathsf{E}} \xrightarrow{\vec{s}} \circ$.

□

This is the main result of this paper:

4.2.3 Corollary *The same statement of Theorem 4.2.2 holds for system* $\mathsf{SBV} \cup \{\circ{\downarrow}\}$ *instead of* BVL.

Proof It follows from Theorems 4.2.2, 2.6 and 3.8. □

The next example shows an application of the marking procedure and the extraction of the computation stepwise from the intermediate proofs. We start with a process structure $[a, \langle a; [\bar{a}, c]\rangle]$ and action sequence $a; c; \circ$. At each step the intermediate proof is obtained by removing marked occurrences and trivial

applications of rules; the associated computation is indicated below:

$$\mathsf{q}\!\downarrow\dfrac{\mathsf{q}\!\downarrow\dfrac{\mathsf{ai}\!\downarrow_{\mathrm{L}}\dfrac{\mathsf{ai}\!\downarrow_{\mathrm{L}}\dfrac{\circ\!\downarrow\dfrac{}{\circ}}{[a,\bar a]}}{\langle[c,\bar c];[a,\bar a]\rangle}}{\langle[a^\bullet,\bar a^\bullet];[c,\bar c];[a,\bar a]\rangle}}{\langle[a^\bullet,\bar a^\bullet];[a,\bar a,c,\bar c]\rangle}}{[a,\langle[a^\bullet,\bar a^\bullet];[\bar a,c,\bar c]\rangle]}\quad\to\quad \mathsf{q}\!\downarrow\dfrac{\mathsf{ai}\!\downarrow_{\mathrm{L}}\dfrac{\circ\!\downarrow\dfrac{}{\circ}}{[a,\bar a]}}{\langle[c^\bullet,\bar c^\bullet];[a,\bar a]\rangle}\quad\to\quad \mathsf{ai}\!\downarrow_{\mathrm{L}}\dfrac{\circ\!\downarrow\dfrac{}{\circ}}{[a^\bullet,\bar a^\bullet]}\quad\to\quad \circ\!\downarrow\dfrac{}{\circ}\quad;$$

$$a.\circ \mid a.(\bar a.\circ \mid c.\circ) \xrightarrow{a} a.\circ \mid \bar a.\circ \mid c.\circ \xrightarrow{c} a.\circ \mid \bar a.\circ \xrightarrow{\circ} \circ \quad .$$

4.3 Comments

Let us summarise the results presented above.

1 Every computation can be put in an easy correspondence to a derivation in SBV, which essentially mimics its behaviour by way of seq and left atomic interaction rules. This result is certainly not unexpected, given that prefixing in CCS is subsumed by the more general form of ordering by seq that we have in SBV.

2 Every proof in SBV$\cup\{\circ\!\downarrow\}$ over a process structure corresponds to a terminating computation. This result is much harder than 1 and it was not obvious. The difficulty, of course, is in the fact that the logical system could perform in principle many more derivations than just those corresponding to computations. It actually does so, but now we know that for each of them there is a terminating computation. The source for the potential applications of this work stems from this result.

The use of point 2, i.e., soundness of SBV with respect to our process algebra, should be the following. BVL, or better yet a further, equivalent restriction along the lines of Miller's uniform proofs, faithfully performs our computations. Here we only have exactly the nondeterminism inherent in the operational semantics of our process algebra. But we can also use the more powerful systems that we know are equivalent to BVL. If we remove the restriction on atomic interactions to be left, as in BV, we can perform communications in any order we like: the time structure of the process is still retained by the logic, but we are not committed to the execution time.

Further, we can add the admissible rule $\mathsf{q}\!\uparrow$: its use allows strongly to limit nondeterminism, so making choices that, if well guided, could reduce dramatically the search space for, say, a verification tool. In addition we can also allow cut rules, in their various forms. These are notoriously extremely effective in reducing exponentially the search space for proofs, provided one knows exactly which structure to use in cuts. As Theorems 2.6 and 3.8 point out, several different systems

are equivalent to BVL. Extending our system to SNEL, an extension of SBV with exponentials studied in [9], will bring in an even larger range of possibilities.

The reader might have noticed that there is little use of the switch rule s when dealing with process structures. This is due to the fact that process structures do not contain copars. The rule s is essential in at least two scenarios:

1 When using the q↑ and cut rules.

2 In the presence of recursion. As I said already, in a coming extension to our system it will be possible to deal with fixpoint constructions. Very briefly, we will deal with structures like $?(\bar{P}, Q)$, which specifies the unlimited possibility of rewriting process P by process Q. For this construct to work, copar and s are essential.

In my opinion, the only really significant challenge remaining in order to capture exactly CCS in a logical system is coping with the silent transition τ. Its algebraic behaviour is rather odd, so I would expect a correspondingly odd logical system, if logical purity is to be maintained. A more sensible approach could be either to give up to perfect correspondence to CCS, or modeling τ by axioms and then studying the impact of this axiomatisation on the properties of interest (cut elimination, mainly).

5 Conclusions

This paper intends to be a contribution to the principled design of logic languages for concurrency. We examined a stripped down version of CCS, having only prefixing and parallel composition, called $\mathsf{PA}_{\mathsf{BV}}$. This very simple process algebra presents a significant challenge to its purely logical account in the proof search paradigm, because of its commutative/non-commutative nature. To the best of my knowledge, the only formal system presenting at the same time commutative, non-commutative and linear operators, necessary to give account of the algebraic nature of $\mathsf{PA}_{\mathsf{BV}}$, is system SBV. Still, there is a nontrivial mismatch, in SBV, between its form of sequentiality and CCS's one.

In this paper I showed how to close this gap, through a purely logical restriction of SBV, and I showed how to represent $\mathsf{PA}_{\mathsf{BV}}$ in SBV. I argued that this process algebra can be extended to a Turing-equivalent one, comprising much of CCS, while still maintaining a perfect correspondence to the purely logical formal system studied in [9]. Further steps, to enhance expressivity, are possible in even more extended formal systems, by way of additives, along the lines of [15].

References

[1] Samson Abramsky and Radha Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543–574, June 1994.

[2] Jean-Marc Andreoli and Remo Pareschi. Linear Objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[3] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

[4] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[5] Alessio Guglielmi. Concurrency and plan generation in a logic programming language with a sequential operator. In P. Van Hentenryck, editor, *Logic Programming, 11th International Conference, S. Margherita Ligure, Italy*, pages 240–254. The MIT Press, 1994.

[6] Alessio Guglielmi. Sequentiality by linear implication and universal quantification. In Jörg Desel, editor, *Structures in Concurrency Theory*, Workshops in Computing, pages 160–174. Springer-Verlag, 1995.

[7] Alessio Guglielmi. A system of interaction and order. Technical Report WV-01-01, Dresden University of Technology, 2001. On the web at: http://www.ki.inf.tu-dresden.de/~guglielm/Research/Gug/Gug.pdf.

[8] Alessio Guglielmi and Lutz Straßburger. Non-commutativity and MELL in the calculus of structures. In L. Fribourg, editor, *CSL 2001*, volume 2142 of *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag, 2001. On the web at: http://www.ki.inf.tu-dresden.de/~guglielm/Research/GugStra/GugStra.pdf.

[9] Alessio Guglielmi and Lutz Straßburger. A non-commutative extension of MELL in the calculus of structures. Technical Report WV-02-03, Dresden University of Technology, 2002. On the web at: http://www.ki.inf.tu-dresden.de/~guglielm/Research/NEL/NELbig.pdf, submitted.

[10] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, May 1994.

[11] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *1992 Workshop on Extensions to Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 242–265. Springer-Verlag, 1993.

[12] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165:201–232, 1996.

[13] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[14] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[15] Lutz Straßburger. A local system for linear logic. Technical Report WV-02-01, Dresden University of Technology, 2002. On the web at: http://www.ki.inf.tu-dresden.de/~lutz/lls.pdf.

[16] Alwen Fernanto Tiu. Properties of a logical system in the calculus of structures. Technical Report WV-01-06, Dresden University of Technology, 2001. On the web at: http://www.cse.psu.edu/~tiu/thesisc.pdf.

Disjunctive Explanations

Katsumi Inoue[1] and Chiaki Sakama[2]

[1] Department of Electrical and Electronics Engineering, Kobe University
Rokkodai, Nada, Kobe 657-8501, Japan
inoue@eedept.kobe-u.ac.jp
[2] Department of Computer and Communication Sciences, Wakayama University
Sakaedani, Wakayama 640-8510, Japan
sakama@sys.wakayama-u.ac.jp

Abstract. Abductive logic programming has been widely used to declaratively specify a variety of problems in AI including updates in data and knowledge bases, belief revision, diagnosis, causal theory, and default reasoning. One of the most significant issues in abductive logic programming is to develop a reasonable method for *knowledge assimilation*, which incorporates obtained explanations into the current knowledge base. This paper offers a solution to this problem by considering *disjunctive explanations* whenever multiple explanations exist. Disjunctive explanations are then to be assimilated into the knowledge base so that the assimilated program preserves all and only minimal answer sets from the collection of all possible updated programs. We describe a new form of abductive logic programming which deals with disjunctive explanations in the framework of *extended abduction*. The proposed framework can be well applied to view updates in disjunctive databases.

1 Introduction

The task of abduction is to infer explanations accounting for an observation. In general, we may encounter multiple explanations for the given observation. When there are multiple explanations of G, we observe that the disjunction of these explanations also accounts for G. In this paper, we formalize this idea by extending the notion of *explanation* to more general one than the traditional framework of *abductive logic programming* (ALP). Suppose that we are given the background knowledge K and a set of abducibles $\mathcal{A}$. Then, each set E of instances of elements from $\mathcal{A}$ satisfying that (i) $K \cup E \models G$ and (ii) $K \cup E$ is consistent, is called an *elementary explanation* in this paper. Then, any disjunction of elementary explanations is called an *explanation*. The reason why we use the term "explanation" for a disjunction of (elementary) explanations is that if $\{e_1\}$ and $\{e_2\}$ are (elementary) explanations of G then, in first-order logic or logic programming with the answer set semantics, $e = e_1 \vee e_2$ satisfies that (i) $K \cup \{e\} \models G$ and (ii) $K \cup \{e\}$ is consistent.

The use of disjunctive explanations is quite natural when the background knowledge K is represented in *disjunctive logic programs*. Also, disjunctive explanations are useful in various applications involving abduction. For example,

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 317–332, 2002.
© Springer-Verlag Berlin Heidelberg 2002

- *Weakest explanations.* In abduction, we usually seek for *least presumptive* or *weakest explanations*. Such an explanation is often called a *weakest sufficient condition* [22]. When $\{e_1\}$ and $\{e_2\}$ are minimal elementary explanations of G, where the minimality is defined in terms of the set inclusion relation, each explanation $\{e_i\}$ $(i = 1, 2)$ is most preferred in traditional formalizations of abduction because $\{e_i\}$ is weaker than any non-minimal explanation like $\{e_1, e_2\}$, i.e., $\{e_1, e_2\} \models e_i$. However, the disjunctive explanation $\{e_1 \vee e_2\}$ is much weaker, i.e., $\{e_i\} \models e_1 \vee e_2$. For another example, when $\{a, b\}$ and $\{c\}$ are the two minimal elementary explanations, $\{a \vee c, b \vee c\}$ is the weakest explanation because we see that $(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$.
- *Skeptical reasoning and minimization.* In query answering from *circumscription* [11], we often need disjunctive explanations. For example, if both $\neg ab(a)$ and $\neg ab(b)$ credulously explain g and the clause $ab(a) \vee ab(b)$ can be entailed from the background theory, then the disjunction $\neg ab(a) \vee \neg ab(b)$ skeptically explains g. A minimization principle with disjunctive explanations is also employed in abduction from *causal theories* [20].
- *Negative (anti-)explanation and contraction of hypotheses.* In *extended abduction* [14], we may want to remove abducible facts from the background theory. For example, suppose that the program is given as:

$$\begin{aligned} g &\leftarrow not\ p, \\ p &\leftarrow a, \\ p &\leftarrow b, \\ a; b\ &, \end{aligned}$$

  and the abducibles are given as $\{a, b\}$. Then, to explain g, it is necessary to remove the disjunction $a; b$ from the program. However, the previous framework of extended abduction [14,13] cannot do that, because only instances of elements from the abducibles can be manipulated. Here, removing $\{a\}$ or $\{b\}$ or $\{a, b\}$ cannot be successful because neither a nor b is in the program.
- *Knowledge base update.* Adapting alternative solutions for an update request to the background theory usually results in multiple alternative new states. The disjunction of these solutions offers a solution representing every possible change in a single state [5,6,25]. This technique reduces the size of knowledge bases through a sequence of updates and keeps only one current knowledge base at a time.

The last application—knowledge base update—is particularly important when we want to assimilate explanations into our current knowledge base. While *knowledge assimilation* is one of the most significant problems in ALP [19,17], not much work has been reported so far. This paper offers a solution to this problem by assimilating disjunctive explanations into a knowledge base. We also introduce disjunctive explanations into the framework of *extended abduction* [14], where both addition and removal of hypotheses are allowed to explain or unexplain an observation. When there are multiple preferred explanations involving removal of hypotheses, assimilating them into one knowledge base is much more difficult than in the case of normal abduction which only adds hypotheses.

It is known that extended abduction can be used to formalize various update problems in AI and databases [14,16,26]. That is, an *insertion/deletion* of a fact G into/from a database is accomplished by a minimal *explanation/anti-explanation* of G. Then, the notion of disjunctive explanations in this paper can also be applied to update problems in databases. In particular, the *view update* problem in *disjunctive databases*, i.e., databases possibly containing disjunctions which represent indefinite or uncertain information, can also be realized within the proposed framework. When we build a database in real-life situations, a database is likely to include such disjunctive facts. Developing an update technique in disjunctive databases is therefore important from practical viewpoints. However, disjunctive databases are more expressive than Datalog [4], and view updates in disjunctive databases are more difficult than the case of Datalog. In fact, there are few studies on the subject of updating disjunctive databases and many problems have been left open. Hence, with our proposed framework, we can make advances in studies of view updates in disjunctive databases.

The rest of this paper is organized as follows. Section 2 reviews a framework of disjunctive logic programs and its answer set semantics. Section 3 introduces the abductive framework considering disjunctive explanations. Section 4 extends our disjunctive abduction to extended abduction which allows removal of abducibles from programs. Section 5 discusses related issues, and Section 6 is a summary. Due to the lack of space, we omit the proofs of theorems in this paper.

2 Disjunctive Programs

A knowledge base or database is represented in an *extended disjunctive program* (EDP) [9], or simply called a *program*, which consists of a finite number of *rules* of the form:

$$L_1; \cdots; L_l \leftarrow L_{l+1}, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n \tag{1}$$

where each L_i is a literal ($n \geq m \geq l \geq 0$), and *not* is *negation as failure* (NAF). The symbol ; represents a disjunction and is often written also as $\vee$. A rule with variables stands for the set of its ground instances. We assume that function symbols never appear in a program, which implies that a number of the ground instances of a variable is finite.[1] The left-hand side of the rule is the *head*, and the right-hand side is the *body*. A rule with the empty head is an *integrity constraint*. Any rule with the empty body $H \leftarrow$ is called a *fact* and is also written as H without the symbol $\leftarrow$.

Any program K is divided into two parts, $K = \mathcal{I}(K) \cup \mathcal{F}(K)$, where $\mathcal{I}(K) \cap \mathcal{F}(K) = \emptyset$, and $\mathcal{I}(K)$ (resp. $\mathcal{F}(K)$) denotes the set of non-fact rules (resp. facts) in K. When we consider a database written as a program, $\mathcal{I}(K)$ (resp. $\mathcal{F}(K)$) represents an *intensional database* (resp. *extensional database*).

[1] This assumption is necessary only for later use in representing explanation closures of an observation in first-order logic (Definition 3.4).

We can consider more general form of programs allowing *nested expressions* [21]. See [21] for the definition of answer sets for such nested programs.[2] An EDP is called an *extended logic program* (ELP) if it contains no disjunction ($l \leq 1$), and an ELP is called a *normal logic program* (NLP) if every L_i is an atom.

The semantics of a program is given by its *answer sets*. First, let K be an EDP without NAF (i.e., $m = n$) and $S \subseteq \mathcal{L}$, where $\mathcal{L}$ is the set of all ground literals in the language of K. Then, S is an *answer set* of K if S is a minimal set satisfying the conditions:

1. For each ground rule $L_1; \cdots; L_l \leftarrow L_{l+1}, \ldots, L_m$ from K, $\{L_{l+1}, \ldots, L_m\} \subseteq S$ implies $\{L_1, \ldots, L_l\} \cap S \neq \emptyset$;
2. If S contains a pair of complementary literals L and $\neg L$, then $S = \mathcal{L}$.

Second, given *any* EDP K (with NAF) and $S \subseteq \mathcal{L}$, consider the EDP (without NAF) K^S obtained as follows: a rule $L_1; \cdots; L_l \leftarrow L_{l+1}, \ldots, L_m$ is in K^S if there is a ground rule of the form (1) from K such that $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$. Then, S is an *answer set* of K if S is an answer set of K^S. An answer set is *consistent* if it is not $\mathcal{L}$. A program is *consistent* if it has a consistent answer set. Note that every answer set S of any EDP is minimal [9], that is, no other answer set S' of K satisfies that $S' \subset S$. The set of all answer sets of K is written as $\mathcal{AS}(K)$. For a literal L, we write $K \models L$ if $L \in S$ for every $S \in \mathcal{AS}(K)$.

3 Disjunctions in Normal Abduction

An *abductive program* is a pair $\langle K, \mathcal{A} \rangle$, where both K and $\mathcal{A}$ are EDPs. Each element of $\mathcal{A}$ and its any instance is called an *abducible*. When a rule is an abducible, it is called an *abducible rule*. Such an abducible rule can be associated with a unique literal called the *name* [12]. Then, with this naming technique, we can always assume in this paper that the abducibles $\mathcal{A}$ of an abductive program $\langle K, \mathcal{A} \rangle$ is a set of literals. Moreover, we assume without loss of generality that, any rule from K having an abducible in its head is always a fact consisting of abducibles only.[3] In abduction, we are given an *observation* G to be explained or unexplained. Without loss of generality, such an observation is assumed to be a non-abducible ground literal [15].

We firstly consider *normal abduction*, and later in Section 4 extend our framework by considering *extended abduction* [14].

[2] Nested expressions are necessary in this paper only because we will later consider the answer sets of a program containing DNF formulas called explanation closures (Theorem 3.2).

[3] A similar assumption is usually used in literature, e.g., [17]. If there is a fact containing both an abducible a and a non-abducible or there is a rule containing an abducible a in its head and a non-empty body, then such an abducible a is made a non-abducible by introducing a rule $a \leftarrow a'$ with a new abducible a' and then replacing a with a' in every fact consisting abducibles only.

Definition 3.1. Let $\langle K, \mathcal{A} \rangle$ be an abductive program and G an observation. A set E is an *elementary explanation of* G *(wrt* $\langle K, \mathcal{A} \rangle$*)* if

1. E is a set of ground instances of elements from $\mathcal{A}$,
2. $K \cup E \models G$, and
3. $K \cup E$ is consistent.

Note here that we use the term "elementary explanation" instead of just calling "explanation". The latter term is reserved for the next definition.

Definition 3.2. Any disjunction of elementary explanations of G is called a *(disjunctive) explanation* of G.

By definition, elementary explanations are also explanations. Disjunctive explanations deserve to be called "explanations" as the next proposition holds.

Proposition 3.1. *Let E be a (disjunctive) explanation of G wrt $\langle K, \mathcal{A} \rangle$. Then, $K \cup E \models G$ and $K \cup E$ is consistent.*

We provide an entailment relationship between programs/explanations as follows. Let R and R' be sets of *formulas with nested expressions* [21]. We write $R \models R'$ if for any $S \in \mathcal{AS}(R)$, there exists $S' \in \mathcal{AS}(R')$ such that $S' \subseteq S$. In this case, we say that R' is *weaker than* R. For example, $\{a, b\} \models \{a\} \models \{a; b\}$. We also say that R and R' are *equivalent* if $\mathcal{AS}(R) = \mathcal{AS}(R')$.

Definition 3.3. An (elementary/disjunctive) explanation E of G is *minimal* (or *weakest*) if for any (elementary/disjunctive) explanation E' of G, $E \models E'$ implies $E' \models E$.

Note that we assumed that the set of abducibles $\mathcal{A}$ consists of literals only. Then, for elementary explanations E and E', the relation $E \models E'$ is equivalent to $E' \subseteq E$. Hence, E is a minimal elementary explanation of G iff no other explanation of G is a proper subset of E.

We can also define an alternative ordering between explanations. Given an abductive program $\langle K, \mathcal{A} \rangle$, we say that an explanation E of G is *less presumptive* than an explanation E' of G if $K \cup E' \models K \cup E$. A *least presumptive explanation* is then defined as a minimal element in the less presumptive relation. We also say that E and E' are *equivalent relative to* K if $\mathcal{AS}(K \cup E) = \mathcal{AS}(K \cup E')$.

Definition 3.4. Let $\mathcal{ME}(G)$ be the set of minimal elementary explanations of G. The *explanation closure of* G *(wrt* $\langle K, \mathcal{A} \rangle$*)* is the disjunctive explanation:

$$\bigvee_{E \in \mathcal{ME}(G)} E\,.$$

The explanation closure gives the least presumptive explanation for the observation. To verify this fact, we consider an alternative formalization of abduction with the enlarged hypothesis space which consists of *disjunctive hypotheses*. Given an abductive program $\langle K, \mathcal{A} \rangle$, the *enlarged abducible set*, written $\mathcal{D}(\mathcal{A})$,

consists of every disjunction of abducibles from $\mathcal{A}$. Then, we can define an abductive program $\langle K, \mathcal{D}(\mathcal{A}) \rangle$, in which we can abduce any disjunction of abducibles to explain an observation. Of course, we can also define elementary and disjunctive explanations for the abductive program $\langle K, \mathcal{D}(\mathcal{A}) \rangle$. However, weakest elementary explanations wrt $\langle K, \mathcal{D}(\mathcal{A}) \rangle$ may contain redundant abducibles as disjuncts. For instance, when K is a program consisting of two rules:

$$\begin{aligned} p &\leftarrow a, \\ &\leftarrow b, \end{aligned}$$

and $\mathcal{A} = \{a, b\}$, as p's explanations, $\{a; b\}$ is weaker than $\{a\}$. To adopt $\{a\}$ as a preferred explanation of p, we need the notion of least presumptive explanations. In this case, $\{a\}$ and $\{a; b\}$ are equivalent relative to K.

Theorem 3.1. *If a formula F is the explanation closure of G wrt $\langle K, \mathcal{A} \rangle$, then F is equivalent (relative to K) to a least presumptive elementary explanation of G wrt $\langle K, \mathcal{D}(\mathcal{A}) \rangle$. Conversely, if E is a least presumptive elementary explanation of G wrt $\langle K, \mathcal{D}(\mathcal{A}) \rangle$, then E is equivalent (relative to K) to the explanation closure of G wrt $\langle K, \mathcal{A} \rangle$.*

Corollary 3.1. *The least presumptive elementary explanation of G wrt $\langle K, \mathcal{D}(\mathcal{A}) \rangle$ is unique up to the equivalence relation relative to K, and is equivalent to the explanation closure of G wrt $\langle K, \mathcal{A} \rangle$.*

Example 3.1. Let K be the program:

$$\begin{aligned} p; \neg q &\leftarrow a,\ b, \\ p &\leftarrow r,\ b, \\ q &\leftarrow c,\ not\, r, \\ r &\leftarrow d,\ not\, q. \end{aligned}$$

Also let the abducibles be $\mathcal{A} = \{a, b, c, d\}$. Then, the minimal elementary explanations of p wrt $\langle K, \mathcal{A} \rangle$ is: $\mathcal{ME}(p) = \{\{a, b, c\}, \{b, d\}\}$. The explanation closure of p is thus

$$F = (a, b, c); (b, d).$$

On the other hand, the least presumptive elementary explanation of p wrt $\langle K, \mathcal{D}(\mathcal{A}) \rangle$ is given by

$$E = \{a; d,\ \ b,\ \ c; d\}.$$

In fact, $\mathcal{AS}(K \cup E) = \mathcal{AS}(K \cup \{F\}) = \{\{a, b, c, p, q\}, \{b, d, p, r\}\}$.

The next theorem states that the explanation closure F of G wrt $\langle K, \mathcal{A} \rangle$ exactly reflects all the possible *minimal changes* from the original program K with the minimal elementary explanations $\mathcal{ME}(G)$ wrt $\langle K, \mathcal{A} \rangle$. With this property, we can say that all possible explanations are *assimilated* into the current program so that the resulting program $K \cup \{F\}$ is uniquely determined. Note here that F is a disjunction of conjunctions of abducibles, that is, a DNF formula. If

necessary, we can convert F into an equivalent CNF formula (by Theorem 3.1) which is in the form of a program. The merit of introduction of explanation closures is that we can just stay in the traditional abductive framework where the abducibles are given as literals and hence it is not necessary to consider the enlarged abducible set for computing weakest explanations.

In the following, for a set $\mathcal{S}$ of sets of literals, we denote the set of minimal elements in $\mathcal{S}$ as $\mu\mathcal{S}$, i.e., $\mu\mathcal{S} = \{\, I \in \mathcal{S} \mid \text{there is no } J \in \mathcal{S} \text{ such that } J \subset I \,\}$.

Theorem 3.2. *Let F be the explanation closure of G wrt $\langle K, \mathcal{A} \rangle$, and $\mathcal{ME}(G)$ be the set of minimal elementary explanations of G wrt $\langle K, \mathcal{A} \rangle$. Then,*

$$\mathcal{AS}(K \cup \{F\}) = \mu \bigcup_{E \in \mathcal{ME}(G)} \mathcal{AS}(K \cup E).$$

Note in Theorem 3.2 that the program augmented with the explanation closure $K \cup \{F\}$ preserves *all and only minimal answer sets* from the collection of programs with individual minimal elementary explanations. In other words, non-minimal answer sets produced by the minimal elementary explanations together with K are lost in $\mathcal{AS}(K \cup \{F\})$. This is because the program $K \cup \{F\}$ is an EDP, of which any answer set is minimal. For example, when the program K is

$$\begin{aligned} a; b\,, \\ p \leftarrow b, \\ p \leftarrow c, \end{aligned}$$

and $\mathcal{A} = \{a, b, c\}$ is the abducibles, we have $\mathcal{ME}(p) = \{b, c\}$. Then,

$$\mathcal{AS}(K \cup \{b\}) \cup \mathcal{AS}(K \cup \{c\}) = \{\{b, p\}, \{a, c, p\}, \{b, c, p\}\}.$$

On the other hand,

$$\mathcal{AS}(K \cup \{b; c\}) = \{\{b, p\}, \{a, c, p\}\}.$$

When we consider the skeptical entailment, non-minimal answer sets are not useful and eliminating them does not change the consequences that are true in all answer sets.

4 Disjunctions in Extended Abduction

In this section, we extend the notion of disjunctive explanations to allow for removal of abducible disjunctions from programs.

We firstly give a definition for *extended abduction* [14,16,26,13]. The following definition is based on [13].

Definition 4.1. Let $\langle K, \mathcal{A} \rangle$ be an abductive program.

1. A pair (P, N) is a *scenario for* $\langle K, \mathcal{A} \rangle$ if P and N are sets of ground instances of elements from $\mathcal{A}$ and $(K \setminus N) \cup P$ is a consistent program.
2. Let G be a ground literal.
   (a) A pair (P, N) is an *elementary explanation of* G *(wrt* $\langle K, \mathcal{A} \rangle$*)* if (P, N) is a scenario for $\langle K, \mathcal{A} \rangle$ such that $(K \setminus N) \cup P \models G$.
   (b) A pair (P, N) is an *elementary anti-explanation of* G *(wrt* $\langle K, \mathcal{A} \rangle$*)* if (P, N) is a scenario for $\langle K, \mathcal{A} \rangle$ such that $(K \setminus N) \cup P \not\models G$.
   (c) An elementary (anti-)explanation (P, N) of G is *minimal* if for any elementary (anti-)explanation (P', N') of G, $P' \subseteq P$ and $N' \subseteq N$ imply $P' = P$ and $N' = N$.

Thus, to explain or unexplain observations, extended abduction not only introduces hypotheses to a program but also removes them from it. On the other hand, abduction in Definition 3.1 is called *normal abduction*, which only introduces hypotheses to explain observations, and is a special case of extended abduction. That is, E is an explanation of G wrt $\langle K, \mathcal{A} \rangle$ (under normal abduction) iff $(E, \emptyset)$ is an explanation of G wrt $\langle K, \mathcal{A} \rangle$ (under extended abduction).

4.1 Problem in Combining Removed Hypotheses

It is not obvious to extend the notion of elementary (anti-)explanations in extended abduction to take disjunctions of multiple (anti-)explanations. The difficulty lies in the following question: when there are more than one way to remove hypotheses in order to (un)explain an observation, how can we construct a combined (anti-)explanations so that the resulting program reflects the semantics for every possible minimal change of the current program? We illustrate this difficulty with the following example.

Example 4.1. [10, Example 3.4] [4] Let K be the program

$$
\begin{aligned}
p &\leftarrow a,\ b,\\
p &\leftarrow e,\\
p &\leftarrow q,\ c,\\
q &\leftarrow a,\ d,\\
a\ &,\\
b;\ d\ &,\\
b;\ e\ &.
\end{aligned}
$$

Suppose that the abducibles are $\mathcal{A} = \{a, b, c, d, e\}$. The unique minimal elementary anti-explanations of p wrt $\langle K, \mathcal{A} \rangle$ is

$$(P_1, N_1) = (\emptyset, \{a\}).$$

[4] Example 4.1 was originally described in the context of *view updates* of disjunctive databases in [10]. Here, we modified it for the use in extended abduction.

On the other hand, there are two minimal elementary anti-explanations of p wrt $\langle K, \mathcal{D}(\mathcal{A}) \rangle$: one is (P_1, N_1), and the other is

$$(P_2, N_2) = (\emptyset, \{b; e\}).$$

To express these two changes in one state, Grant *et al.* [10] actually construct the two programs by reflecting these two anti-explanations on the fact part $\mathcal{F}(K)$:

$$K_1 = \mathcal{I}(K) \cup \{\, b; d, \;\; b; e \,\},$$
$$K_2 = \mathcal{I}(K) \cup \{\, a, \;\; b; d \,\}.$$

Then, [10] takes the disjunction of these fact parts, i.e., $\mathcal{F}(K_1) \vee \mathcal{F}(K_2)$, and converting the resulting DNF formula into CNF, yielding

$$((b \vee d) \wedge (b \vee e)) \vee (a \wedge (b \vee d)) = (b \vee d) \wedge (a \vee b \vee e).$$

That is, the new program is computed as

$$K' = \mathcal{I}(K) \cup \{\, b; d, \;\; a; b; e \,\}.$$

By computing the difference between K and K', an anti-explanation of p would be expressed as

$$(P', N') = (\{\, a; b; e \,\}, \; \{\, a, \; b; e \,\}).$$

Unless we follow this expensive procedure, it is difficult to compose the last scenario (P', N') directly from the minimal elementary anti-explanations, (P_1, N_1) and (P_2, N_2), of p wrt $\langle K, \mathcal{D}(\mathcal{A}) \rangle$. Moreover, it is impossible to construct (P', N') only from the unique minimal elementary explanation (P_1, N_1) of p wrt $\langle K, \mathcal{A} \rangle$.

From the above example, one may expect that two (anti-)explanations, (P_1, N_1) and (P_2, N_2), can be combined by constructing a new (anti-)explanation:

$$(\{P_1 \vee P_2, \;\; N_1 \vee N_2\}, \;\; N_1 \cup N_2).$$

Unfortunately, this is not the case as the next example shows.

Example 4.2. Let K be the program

$$p \leftarrow a, \; not\, b,$$
$$p \leftarrow a, \; not\, c,$$
$$b,$$
$$c,$$

and the abducibles be $\mathcal{A} = \{a, b, c\}$. The two minimal elementary explanations of p is $(\{a\}, \{b\})$ and $(\{a\}, \{c\})$. Combining these two in the above way results in $(P, N) = (\{\, a, \; b; c \,\}, \; \{\, b, \; c \,\})$. However, this scenario cannot be an explanation of p because $(K \setminus N) \cup P \not\models p$.

4.2 From Extended Abduction to Normal Abduction

From the discussion in Section 4.1, we had better consider an alternative way to combine multiple (anti-)explanations in extended abduction. In [13], extended abduction is shown to be reduced to normal abduction. Here, we use this method to translate removal of abducibles from programs to addition of abducibles to programs. Recall that, without loss of generality, the set of abducibles $\mathcal{A}$ can be assumed to be a set of literals and there is no rule which has a non-empty body and a head containing abducible literals. Under this assumption, the translation ν shown in [13] is simplified as follows. For addition of an abducible literal, we do not have to give it a name and leave it as it is. For removal of an abducible literal a, we give a name to a through NAF by *not* $del(a)$. Then, deletion of an abducible a is realized by addition of $del(a)$ to the program.

For an abductive program $\langle K, \mathcal{A} \rangle$, the program $\nu(K, \mathcal{A}) = \langle \nu(K), \nu(\mathcal{A}) \rangle$ is defined as follows.

$$\nu(K) = (K \setminus \mathcal{A}) \cup \{\, a \leftarrow not\ del(a) \mid a \in K \cap \mathcal{A} \,\},$$
$$\nu(\mathcal{A}) = \mathcal{A} \cup \{\, del(a) \mid a \in K \cap \mathcal{A} \,\}.$$

Theorem 4.1. [13, Theorem 1] (P, N) is a minimal elementary explanation of G wrt $\langle K, \mathcal{A} \rangle$ under extended abduction iff E is a minimal elementary explanation of G wrt $\nu(K, \mathcal{A})$ under normal abduction, where $P = \{a \mid a \in E \cap \mathcal{A}\}$ and $N = \{a \mid del(a) \in E\}$.

The above theorem presents that all minimal elementary explanations are computable by normal abduction from $\nu(K, \mathcal{A})$. For anti-explanations, the next theorem shows that $\nu(K, \mathcal{A})$ preserves every minimal elementary anti-explanation of $\langle K, \mathcal{A} \rangle$ in the form of a scenario $(E, \emptyset)$. Namely, we do not have to consider removal of hypotheses in a scenario. Then, to compute these anti-explanations, we can utilize the relationship between explanations and anti-explanations (see [13, Theorem 2]).

Theorem 4.2. (P, N) *is a minimal elementary anti-explanation of* G *wrt* $\langle K, \mathcal{A} \rangle$ *iff* $(E, \emptyset)$ *is a minimal anti-explanation of* G *wrt* $\nu(K, \mathcal{A})$*, where* $P = \{a \mid a \in E \cap \mathcal{A}\}$ *and* $N = \{a \mid del(a) \in E\}$.

4.3 Disjunctive (Anti-)Explanations

Now, we are ready to compose disjunctive explanations for extended abduction. Firstly, we extend Definition 4.1 for extended abduction by allowing removal of disjunctive hypotheses from a program.

Definition 4.2. Let $\langle K, \mathcal{A} \rangle$ be an abductive program, G a ground literal.

1. A pair (P, N) is a *d-scenario for* $\langle K, \mathcal{A} \rangle$ if P is a set of ground instances of elements from $\mathcal{A}$ and N is a set of ground instances of elements from $\mathcal{D}(\mathcal{A})$ such that $(K \setminus N) \cup P$ is a consistent.

2. A d-scenario (P, N) is an *elementary d-explanation of* G *(wrt* $\langle K, \mathcal{A} \rangle$*)* if $(K \setminus N) \cup P \models G$.
3. A d-scenario (P, N) is an *elementary d-anti-explanation of* G *(wrt* $\langle K, \mathcal{A} \rangle$*)* if $(K \setminus N) \cup P \not\models G$.
4. An elementary d-(anti-)explanation (P, N) of G is *minimal* if for any elementary d-(anti-)explanation (P', N') of G, $P \models P'$ and $N \models N'$ imply $P' \models P$ and $N' \models N$.

In the above definition, we allow removal of disjunctive hypotheses from the enlarged abducible set $\mathcal{D}(\mathcal{A})$, but addition of hypotheses is allowed only from the literal abducibles $\mathcal{A}$. This asymmetry is due to our intention that hypotheses to be added should be made disjunctive just in the same way as normal abduction although hypotheses to be removed could only be translated into normal abduction through NAF of the form $not\ del(_)$. Note also that the minimality of d-(anti-)explanations is now defined through the entailment relationship.

For translating abducible removal into abducible addition, we slightly modify the mapping ν for preserving minimal elementary (anti-)explanations, and consider the mapping ν^d as follows. For an abductive program $\langle K, \mathcal{A} \rangle$, the program $\nu^d(K, \mathcal{A}) = \langle \nu^d(K), \nu^d(\mathcal{A}) \rangle$ is defined as follows.

$$\nu^d(K) = (K \setminus \mathcal{D}(\mathcal{A})) \cup \{ a \leftarrow not\ del(a) \mid a \in K \cap \mathcal{D}(\mathcal{A}) \},$$
$$\nu^d(\mathcal{A}) = \mathcal{A} \cup \{ del(a) \mid a \in K \cap \mathcal{D}(\mathcal{A}) \}.$$

Note that the difference between ν and ν^d is that the naming technique is applied to the enlarged abducible set $\mathcal{D}(\mathcal{A})$ instead of the original abducibles $\mathcal{A}$ only. The new abducible set $\nu^d(\mathcal{A})$ is, however, defined with $\mathcal{A}$ without considering disjunctive hypotheses. This is because we do not have to consider any removal of hypotheses for $\nu^d(K, \mathcal{A})$ so that we can define the notions of *(disjunctive) explanations*, *minimal explanations*, and *explanation closures* in the same way as Definitions 3.2, 3.3, and 3.4 for normal abduction. Similarly, we can define the closure formula for anti-explanations as follows.

Definition 4.3. The *anti-explanation closure of* G *(wrt* $\langle K, \mathcal{A} \rangle$*)* is the disjunctive explanation:

$$\bigvee_{(E,\emptyset) \in \mathcal{MEA}^{\nu}(G)} E,$$

where $\mathcal{MEA}^{\nu}(G)$ is the set of all minimal elementary anti-explanations of G wrt $\nu^d(K, \mathcal{A})$.

The following theorems show that the translation ν^d preserves the minimal answer sets from the program augmented with any minimal elementary d(-anti)-explanation. Here, for a program K containing literals of the form $del(_)$, we will write:

$$\mathcal{AS}^{-del}(K) = \mu \{ S \cap \mathcal{L}_K \mid S \in \mathcal{AS}(K) \},$$

where $\mathcal{L}_K$ denotes the set of literals in the language of K not containing any literal of the form $del(_)$. Note that we need to select the minimal elements from

the right hand side. This is because eliminating all literals of the form $del(_)$ from each answer set may produce a literal set that properly includes others.

Theorem 4.3. *Let F be the explanation closure of G wrt $\langle K, \mathcal{A} \rangle$, and $\mathcal{ME}^d(G)$ be the set of minimal elementary d-explanations of G wrt $\langle K, \mathcal{A} \rangle$. Then,*

$$\mathcal{AS}^{-del}(\nu^d(K) \cup \{F\}) = \mu \bigcup_{(P,N) \in \mathcal{ME}^d(G)} \mathcal{AS}((K \setminus N) \cup P).$$

Theorem 4.4. *Let H be the anti-explanation closure of G wrt $\langle K, \mathcal{A} \rangle$, and $\mathcal{MEA}^d(G)$ be the set of minimal elementary d-anti-explanations of G wrt $\langle K, \mathcal{A} \rangle$. Then,*

$$\mathcal{AS}^{-del}(\nu^d(K) \cup \{H\}) = \mu \bigcup_{(P,N) \in \mathcal{MEA}^d(G)} \mathcal{AS}((K \setminus N) \cup P).$$

Example 4.3. (cont. from Example 4.1) The fact part $\mathcal{I}(K) = K \cap \mathcal{D}(\mathcal{A}) = \{ a,\ b; d,\ b; e \}$ is translated into

$$a \leftarrow not\ del(a),$$
$$b; d \leftarrow not\ del(b; d),$$
$$b; e \leftarrow not\ del(b; e).$$

The two minimal elementary anti-explanations of p wrt $\nu^d(K, \mathcal{A})$ are $(\{del(a)\}, \emptyset)$ and $(\{del(b; e)\}, \emptyset)$, which respectively correspond to the two d-anti-explanations of p wrt $\langle K, \mathcal{A} \rangle$, $(P_1, N_1) = (\emptyset, \{a\})$ and $(P_2, N_2) = (\emptyset, \{b; e\})$. Then, the anti-explanation closure of p is $H = del(a); del(b; e)$. Assimilating this formula into the program, we obtain the new program $K' = \nu^d(K) \cup \{del(a); del(b; e)\}$. Then,

$$\mathcal{AS}^{-del}(K') = \{\{b\}, \{d, e, p\}, \{a, d, q\}\}.$$

Example 4.4. (cont. from Example 4.2) The fact part $\mathcal{F}(K)$ is translated into

$$b \leftarrow not\ del(b),$$
$$c \leftarrow not\ del(c).$$

The two minimal elementary explanations of p wrt $\nu^d(K, \mathcal{A})$ are $\{a, del(b)\}$ and $\{a, del(c)\}$, which respectively correspond to $(\{a\}, \{b\})$ and $(\{a\}, \{c\})$. Then, the explanation closure is $F = (a, del(b)); (a, del(c))$. By converting F into CNF, the minimal explanation of p wrt $\nu^d(K, \mathcal{A})$ is obtained as $E = \{ a,\ del(b); del(c) \}$. Then,

$$\mathcal{AS}^{-del}(\nu^d(K) \cup E) = \{\{a, b, p\}, \{a, c, p\}\}.$$

5 Related Work

1. Disjunctive explanations. The idea of taking a disjunction of multiple explanations has appeared at times in the literature of computing abduction, although

no previous work has formally investigated the effect of such disjunctive explanations in depth. Helft *et al.* [11] define an explanation as a disjunction of elementary explanations in abduction from first-order theories for answering queries in *circumscription*. Konolige [20] defines a *cautious explanation* as a disjunction of all preferred explanations, and uses it to relate consistency-based explanations with abductive explanations in propositional causal theories. Lin [22] provides a method to compute *weakest sufficient conditions* for propositional theories, in which he constructs the disjunction of elementary explanations obtained from prime implicates. In ALP, disjunctions of elementary explanations are sometimes obtained in computing abduction through Clark completion [3,8,23]. Such procedures are designed for computing normal abduction from hierarchical or acyclic NLPs. Inoue and Sakama [16] extend this completion method to compute extended abduction. We can use these procedures to compute explanation closures directly in some restricted classes of logic programs.

2. View updates in disjunctive databases. Although there are some studies on updating incomplete information in relational databases [1], only a few works [10,7] focused on updating disjunctive databases. Grant *et al.* [10] translate view updates into a set of disjunctive facts based on expansion of an SLD-tree, so that updates are achieved by inserting/deleting these disjunctive facts to/from a database. Their method is correct for stratified programs, but cannot achieve an insertion of p into a non-stratified EDP K shown in Example 3.1. Fernández *et al.* [7] realize view updates in a wide class of EDPs through construction of minimal models that satisfy an update request. In their algorithm, however, computation is done on all possible models of the Herbrand base, and how to compute disjunctive solutions directly from changes of facts was an *open problem* in the class of EDPs. We solved this problem by translating extended abduction to normal abduction without computing all possible models. Furthermore, updates are performed without using abduction in [10,7]. Hence, the notion of disjunctive (anti-)explanations in abduction does not appear in these work.

For non-disjunctive deductive databases, abductive frameworks have been used to realize view updates. Bry [2] translates abduction into a disjunctive program and database updates are realized by bottom-up computation on a meta-program specifying an update procedure. Kakas and Mancarella [18] characterize view updates through abduction in deductive databases. The procedures in [18,2] are based on normal abduction and do not consider extended abduction.

3. Knowledge assimilation with abduction. Not much work has been reported to assimilate obtained multiple explanations into the current knowledge base. Kakas and Mancarella [19] discussed two ways for handling the problem of multiple explanations. One is to generate all consistent scenarios accounting for an observation and work with all of them simultaneously. They suggest to use an ATMS for this purpose. The other way is to generate one preferred explanation at a time according to some priority. Since such a choice of explanation could be wrong in the subsequent observations, they suggest the use of a belief revision mechanism through a Doyle-style TMS.

Our proposal somewhat differs from Kakas and Mancarella's two methods. Our method is similar to the spirit suggested by Fagin *et al.* [5], which defines the result of assimilation or updates to be the disjunction of all the possible theories with minimal change. This method presents a semantically consistent picture of theory changes. Rossi and Naqvi [25] optimize this approach by taking the disjunction of updated extensional databases instead of composing the disjunction of the whole databases with intensional ones. Grant *et al.* [10] follow the same line on view updates in disjunctive databases. An interesting alternative approach is also suggested by Fagin *et al.* [6], in which multiple alternative theories called "flocks" are kept as they are.

6 Summary

This paper has presented a method to construct the weakest explanations and anti-explanations in normal and extended abduction. For normal abduction, we formally established the effect of disjunctive explanations, in which all and only minimal answer sets are preserved for the minimal elementary explanations. We also presented that the explanation closure is equivalent to a least presumptive explanation consisting of disjunctive hypotheses. These results imply a practical merit that computing least presumptive explanations wrt $\langle K, \mathcal{D}(\mathcal{A}) \rangle$ can easily be realized by traditional abductive procedures [18,3,15,8,17,16] for $\langle K, \mathcal{A} \rangle$ or corresponding *answer set programming* [24] which simulates normal abduction. That is, the minimal elementary explanations are firstly computed by these procedures, then the disjunction of them is just composed. We have also applied these results to extended abduction, and proposed a method to combine multiple solutions that involve removal of hypotheses.

The notion of disjunctive explanations is quite useful in various applications, and our method has shed some light on the problem of knowledge assimilation. In particular, considering view updates in disjunctive databases is generally difficult in the presence of disjunctive information. Our solution in this paper correctly achieves view updates in a large class of disjunctive databases.

References

1. S. Abiteboul. Updates, a new frontier. In: *Proceedings of the 2nd International Conference on Database Theory, Lecture Notes in Computer Science*, 326, Springer, pages 1–18, 1988. 329
2. F. Bry. Intensional updates: abduction via deduction. In: *Proceedings of ICLP '90*, pages 561–575, MIT Press, 1990. 329
3. L. Console, D. Theseider Dupré and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1:661–690, 1991. 329, 330
4. T. Eiter, G. Gottlob, and H. Mannila. Adding disjunction to Datalog. In: *Proc. 13th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 267–278, 1994. 319

5. R. Fagin, J. D. Ullman, and M. Y. Vardi. On the semantics of updates in databases (preliminary report). In: *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 352–365, 1983. 318, 330
6. R. Fagin, G. M. Kuper, J. D. Ullman, and M. Y. Vardi. Updating logical databases. In: *Advances in Computing Research*, Volume 3, pages 1–18, JAI Press, 1986. 318, 330
7. J. Fernández, J. Grant and J. Minker. Model theoretic approach to view updates in deductive databases. *Journal of Automated Reasoning*, 17:171–197, 1996. 329
8. T. H. Fung and R. Kowalski. The iff procedure for abductive logic programming. *Journal of Logic Programming*, 33:151–165, 1997. 329, 330
9. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991. 319, 320
10. J. Grant, J. Horty, J. Lobo and J. Minker. View updates in stratified disjunctive databases. *Journal of Automated Reasoning*, 11:249–267, 1993. 324, 325, 329, 330
11. N. Helft, K. Inoue, and D. Poole. Query answering in circumscription. In: *Proceedings of IJCAI-91*, pages 426–431, Morgan Kaufmann, 1991. 318, 329
12. K. Inoue. Hypothetical reasoning in logic programs. *Journal of Logic Programming*, 18(3):191-227, 1994. 320
13. K. Inoue. A simple characterization of extended abduction. In *Proceedings of the 1st International Conference on Computational Logic, Lecture Notes in Artificial Intelligence*, 1861, pages 718–732, Springer, 2000. 318, 323, 326
14. K. Inoue and C. Sakama. Abductive framework for nonmonotonic theory change. In: *Proceedings of IJCAI-95*, pages 204–210, Morgan Kaufmann, 1995. 318, 319, 320, 323
15. K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996. 320, 330
16. K. Inoue and C. Sakama. Computing extended abduction through transaction programs. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):339-367, 1999. 319, 323, 329, 330
17. A. C. Kakas, R. A. Kowalski and F. Toni. The role of abduction in logic programming. In: D. M. Gabbay, C. J. Hogger and J. A. Robinson (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324, Oxford University Press, 1998. 318, 320, 330
18. A. C. Kakas and P. Mancarella. Database updates through abduction. In: *Proceedings of the 16th International Conference on Very Large Databases*, pages 650–661, Morgan Kaufmann, 1990. 329, 330
19. A. C. Kakas and P. Mancarella. Knowledge assimilation and abduction. In: J. P. Martins and M. Reinfrank (eds.), *Truth Maintenance Systems, Lecture Notes in Artificial Intelligence*, 515, 54–70, Springer, 1991. 318, 329
20. K. Konolige. Abduction versus closure in causal theories. *Artificial Intelligence*, 53:255–272, 1992. 318, 329
21. V. Lifschitz, L. R. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999. 320, 321
22. F. Lin. On strongest necessary and weakest sufficient conditions. In: *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning*, pages 167–175, Morgan Kaufmann, 2000. 318, 329
23. F. Lin and J.-H. You. Abductive logic programming: a new definition and an abductive procedure based on rewriting. In: *Proceedings of IJCAI-01*, pages 655–661, Morgan Kaufmann, 2001. 329

24. V. W. Marek, and M. Truszczyński. Stable models and an alternative logic programming paradigm. In: K. R. Apt *et al.*, editors, *The Logic Programming Paradigm—A 25 Year Perspective*, pages 375–398, Springer, 1999. 330
25. F. Rossi and S. A. Naqvi. Contributions to the view update problem. In: *Proceedings of ICLP '89*, pages 398–415, MIT Press, 1989. 318, 330
26. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In: *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, *Lecture Notes in Artificial Intelligence*, 1730, pages 147-161, Springer, 1999. 319, 323

Reasoning with Infinite Stable Models II: Disjunctive Programs

Piero A. Bonatti

Dip. di Tecnologie dell'Informazione – Università di Milano
I-26013 Crema, Italy
bonatti@dti.unimi.it

Abstract. The class of *finitary normal logic programs*—identified recently, in [1]—makes it possible to reason effectively with function symbols, recursion, and infinite stable models. These features may lead to a full integration of the standard logic programming paradigm with the answer set programming paradigm. For all finitary programs, ground goals are decidable, while nonground goals are semidecidable. Moreover, the existing engines (that currently accept only much more restricted programs [11,7]) can be extended to handle finitary programs by replacing their front-ends and keeping their core inference mechanism unchanged. In this paper, the theory of finitary normal programs is extended to disjunctive programs. More precisely, we introduce a suitable generalization of the notion of finitary program and extend all the results of [1] to this class. For this purpose, a consistency result by Fages is extended from normal programs to disjunctive programs. We also correct an error occurring in [1].

1 Introduction

For a long time—in the framework of the stable model semantics—function symbols, recursive data structures and recursion have been believed to lie beyond the threshold of computability. Only recently, in [1], the class of so-called *finitary normal programs* has been identified, that makes it possible to reason effectively with normal logic programs with function symbols, recursion, and infinite stable models. Ground goals are decidable, while nonground goals are semidecidable. The latter can simulate the computations of arbitrary Turing machines. A nice property of finitary programs is that the existing engines (that currently accept only much more restricted programs [11,7]) can be extended to handle finitary programs by replacing their front-ends and keeping their core inference mechanism unchanged.

The role of the front-end is building a fragment of the program's ground instantiation, relevant to the given query. In this stage, resolution-based and top-down partial evaluation techniques may come into play. Subsequently, the standard problem solvers for reasoning under the stable model semantics can be applied to the selected fragment. In this way, the standard logic programming paradigm and the answer set programming paradigm can be effectively

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 333–347, 2002.
© Springer-Verlag Berlin Heidelberg 2002

integrated. Such techniques are useful both for extending the expressiveness of the two paradigms, and for tackling larger problems, e.g., in the area of planning, where the size of the Herbrand universe easily exceeds the memory of the existing answer set computation engines.

Recognizing finitary programs is, in general, an undecidable problem. However, there exist prototype tools, based on static program analysis techniques, that are able to recognize a large and powerful (Turing equivalent) class of finitary programs. These tools have been demonstrated at the LPNMR'01 conference and are described in [2].

In this paper, the theory of finitary normal programs is extended to disjunctive programs. More precisely, we introduce a suitable generalization of the notion of finitary programs and extend all the results of [1] to this class. For this purpose, a consistency result by Fages is extended from normal programs to disjunctive programs. We also correct an error occurring in [1]. This work opens the way to extending inference engines for disjunctive programs with negation (such as DLV [7]), by making them capable of handling function symbols and recursion.

The paper is organized as follows. After some preliminaries (Section 2), we extend Fages' consistency theorem for normal programs in Section 3. Disjunctive finitary programs are introduced in Section 4, and their properties are illustrated in the same section. The paper is closed by some conclusions (Section 5).

2 Preliminaries

Disjunctive logic programs (disjunctive programs, for short) are sets of rules of the form $A_1 \vee \ldots \vee A_m \leftarrow L_1, \ldots, L_n$ $(m > 0,\ n \geq 0)$ where each A_i is a logical atom and each L_i $(i = 1, \ldots n)$ is a literal. If R is a rule with the above form, let $head(R) = \{A_1, \ldots, A_m\}$, $body(R) = \{L_1, \ldots, L_n\}$. A program P is *normal* if for all $R \in P$, $|head(R)| = 1$. The ground instantiation of a program P is denoted by $Ground(P)$. The *Gelfond-Lifschitz transformation* P^I of program P w.r.t. an Herbrand interpretation I (represented as usual, as a set of ground atoms) is obtained by removing from $Ground(P)$ all the rules containing a negative literal $\neg B$ such that $B \in I$, and by removing from the remaining rules all negative literals. An interpretation M is a *stable model* of P if M is a minimal Herbrand model of P^M. A formula F is *credulously* (resp. *skeptically*) *entailed* by P iff F is satisfied by some (resp. each) stable model of P.[1]

The *atom dependency graph* (or simply *dependency graph*) of a program P is a labelled directed graph denoted by $DG(P)$, whose vertices are the ground atoms of P's language. Moreover, *i)* there exists an edge labelled '+' (called positive edge) from A to B iff for some rule $R \in Ground(P)$, $A \in head(R)$ and $B \in body(R)$; *ii)* there exists an edge labelled '−' (called negative edge) from A to B iff for some rule $R \in Ground(P)$, $A \in head(R)$ and $\neg B \in body(R)$.

[1] Here by "formula" we mean any classical formula. Accordingly, satisfaction is classical satisfaction.

An atom A depends positively (resp. negatively) on B if there is a directed path from A to B in the dependency graph with an even (resp. odd) number of negative edges. Moreover, each atom depends positively on itself. If A depends positively (resp. negatively) on B we write $A \geq_+ B$ (resp. $A \geq_- B$). We write $A \geq B$ if either $A \geq_+ B$ or $A \geq_- B$. If $A \geq B$ and $B \not\geq A$, then we write $A > B$. If both $A \geq_+ B$ and $A \geq_- B$ hold then we write $A \geq_\pm B$. Relation $\geq$ induces an equivalence relation as follows: $A \sim B$ iff $A \geq B$ and $B \geq A$. Superscript P will be added to the above relations (e.g., as in $A \geq_+^P B$) whenever the program P whose dependency graph induces the relations is not clearly identified by the context.

By *odd-cycle* we mean a cycle in the dependency graph with an odd number of negative edges. A ground atom is *odd-cyclic* if it occurs in an odd-cycle. A program is *order consistent* if there are no infinite chains $A_1 \geq_\pm A_2 \geq_\pm \ldots \geq_\pm A_i \geq_\pm \ldots$ (note that odd-cycles are a special case of such chains, where each atom occurs infinitely often).

Theorem 1 (Fages [8]). *Every order consistent, normal logic program has at least one stable model.*

A *splitting set* for a program P [10] is a set of atoms U closed under the following property: for all rules $R \in Ground(P)$, if $head(R) \cap U \not= \emptyset$ then U contains all the atoms occurring in R. We call a literal whose atom belongs to U an *U-literal*. The set of rules $R \in Ground(P)$ with $head(R) \cap U \not= \emptyset$—called the *bottom* of P w.r.t. U—will be denoted by $b_U(P)$. By $e_U(P, I)$, where I is a Herbrand interpretation, we denote the following partial evaluation of P w.r.t. $I \cap U$: remove from $Ground(P)$ each rule R such that some U-literal $L_i \in body(R)$ is false in I, and remove from the remaining rules all the U-literals L_i.

Theorem 2 (Splitting theorem [10]). *Let U be a splitting set for a disjunctive logic program P. An interpretation M is a stable model of P iff $M = J \cup I$, where*

1. *I is a stable model of $b_U(P)$, and*
2. *J is a stable model of $e_U(Ground(P) \setminus b_U(P), I)$.*

A *normal shift* of a disjunctive program P is a normal program P^s obtained from P by replacing each rule of the form $A_1 \vee \ldots \vee A_m \leftarrow L_1, \ldots, L_n$ with one rule of the form $A_i \leftarrow L_1, \ldots, L_n, \neg A_1, \ldots, \neg A_{i-1}, \neg A_{i+1} \ldots, \neg A_m$ (for some $1 \leq i \leq m$).

Theorem 3 ([3]). *If P^s is a (normal) shift of P, then all the stable models of P^s are also stable models of P.*

Next we recall the basics of finitary normal programs.

Definition 1 (Finitary programs). *We say a normal logic program P is* finitary *if the following conditions hold:*

1. *For each node A in the dependency graph of P, the set of nodes $\{B \mid A \geq B\}$ is finite.*
2. *Only a finite number of nodes of the dependency graph of P occurs in an odd-cycle.* □

For example, most classical programs on recursive data structures such as lists and trees (e.g. predicates `member`, `append`, `reverse`) satisfy the first condition. In these programs, the terms occurring in the body of a rule occur also in the head, typically as strict subterms of the head's arguments. This property clearly entails Condition 1.

The second condition is satisfied by most of the programs used for embedding NP-hard problems into logic programs [4,6,9]. Such programs can be (re)formulated by using a single odd cycle involving one atom p and defined by simple rules such as $p \leftarrow \neg p$ and $p \leftarrow f, \neg p$ (if p does not occur elsewhere, then f can be used as the logical constant *false* in the rest of the program).

An example of finitary program without odd-cycles is illustrated in Figure 1. It credulously entails a ground goal $\mathtt{s}(t)$ iff t encodes a satisfiable formula. By adding rule $\bot \leftarrow \neg\mathtt{s}(f), \neg\bot$ we obtain another finitary program with one odd-cycle, such that $\mathtt{s}(t)$ is skeptically entailed iff the formula encoded by t is a logical consequence of the one encoded by f.

$\mathtt{s(and(X,Y))} \leftarrow \mathtt{s(X)}, \mathtt{s(Y)}$	$\mathtt{s(A)} \leftarrow \mathtt{member(A,[p,q,r,s])}, \neg\mathtt{ns(A)}$
$\mathtt{s(or(X,Y))} \leftarrow \mathtt{s(X)}$	$\mathtt{ns(A)} \leftarrow \mathtt{member(A,[p,q,r,s])}, \neg\mathtt{s(A)}$
$\mathtt{s(or(X,Y))} \leftarrow \mathtt{s(Y)}$	$\mathtt{member(A,[A|L])}$
$\mathtt{s(not(X))} \leftarrow \neg\mathtt{s(X)}$	$\mathtt{member(A,[B|L])} \leftarrow \mathtt{member(A,L)}$

Fig. 1. A finitary program for SAT

3 Extending Fages' Theorem

This section focusses on programs that satisfy the first condition defining finitary programs [1]. We call such programs *finitely recursive*.

Definition 2. *A disjunctive program P is* finitely recursive *iff for all ground atoms A in the language of P, the set $\{B \mid A \geq B\}$ is finite.*

Informally speaking, and from a backward chaining perspective, the predicates defined by finitely recursive programs can fall into an infinite loop, but only if the loop consists in a finite cycle (as opposed to more general infinite sequences of subgoals).

We conjecture that all order-consistent and finitely-recursive disjunctive programs have at least one stable model. However, we currently have no formal proof of this conjecture, and must leave its demonstration as an open problem.

In this section we prove a weaker result, by combining Fages' consistency theorem for normal logic programs (Theorem 1), and a property of shifts (Theorem 3).

Our first consistency result is a direct consequence of Theorem 1 and Theorem 3.

Lemma 1. *Let P^s be a normal shift of P. If P^s is order consistent then P has at least one stable model.*

Example 1. Let P consist of the rules

$$R_1 = p \vee q \vee r \leftarrow \neg r$$
$$R_2 = s \vee t \leftarrow \neg p$$
$$R_3 = s \leftarrow \neg q$$
$$R_4 = q \leftarrow t$$
$$R_5 = t \leftarrow \neg z$$
$$R_6 = z \leftarrow s.$$

P has the following order consistent normal shift, whose unique stable model is $\{q, s, z\}$.

$$R'_1 = q \leftarrow \neg p, \neg r, \neg r$$
$$R'_2 = s \leftarrow \neg t, \neg p$$
$$R_3 = s \leftarrow \neg q$$
$$R_4 = q \leftarrow t$$
$$R_5 = t \leftarrow \neg z$$
$$R_6 = z \leftarrow s.$$

Next, we identify sufficient conditions for the existence of an order consistent normal shift P^s. In the following, for all sets of ground atoms S, we denote by $\max_{\geq}(S)$ the set of all $A \in S$ such that for no $B \in S$, $B > A$.

Definition 3. *A disjunctive program P is* shift consistent *if the following conditions hold:*

1. *P is order consistent.*
2. *For all $R \in Ground(P)$ and all distinct A and B in $\max_{\geq}(head(R))$, if $A \sim B$ then $A \geq_- B$.*

Example 2. In Example 1, $\max_{\geq}(head(R_1)) = \{q\}$, so the second condition of Definition 3 is vacuously true. Moreover, $\max_{\geq}(head(R_2)) = \{s, t\}$, and the only dependencies between these two atoms are $s \geq_- t$ and $t \geq_- s$. Finally, P is order consistent because its Herbrand domain is finite and the dependency graph contains no odd-cycles. It follows that P is shift consistent.

Lemma 2. *All finitely recursive and shift consistent disjunctive programs P have an order consistent normal shift P^s.*

From Lemma 2 and Lemma 1 we immediately obtain our main consistency result.

Theorem 4. *All finitely recursive and shift consistent disjunctive programs P have at least one stable model.*

The second shift consistency condition has some strong consequences, that restrict the form of rule heads.

Proposition 1. *Let P be a shift consistent program, and let R be any rule in P. For all equivalence classes C in the quotient set $\max_{\geq}(head(R))/\sim$,*

1. $|C| \leq 2$.
2. *If $C = \{A, B\}$ and $A \not=B$, then $A \not\geqq B$ and $B \not\geqq A$.*

Note that the constraint $|C| \leq 2$ does not mean that the entire head must contain no more than two atoms.

Example 3. Let P be

$$R_1 = p \vee q \vee r$$
$$R_2 = p \leftarrow \neg q$$
$$R_3 = q \leftarrow \neg p.$$

Here $\max_{\geq}(head(R_1))/\sim = \{\{p, q\}, \{r\}\}$. Both equivalence classes have at most two atoms. It is not hard to see that this program is shift consistent.

3.1 Extensions and Refinements

The second condition of Definition 3 (and the corresponding restrictions stated in Proposition 1), can be relaxed without affecting the consistency result. For this purpose, we need a notion of dependency linearization. Intuitively, the linearization process completes the partial preorder $\geq$ without introducing new equivalences.

Definition 4. *A* linearization *of $\geq$ is a preorder $\succeq$ over the set of ground atoms that includes $\geq$, and such that*

1. *for all ground atoms A and B, either $A \succeq B$ or $B \succeq A$.*
2. *$A \succeq B$ and $B \succeq A$ hold simultaneously only if $A \sim B$.*

Example 4. Consider again Example 3. There, $p \not\geq r$, $r \not\geq p$, $q \not\geq r$, and $r \not\geq q$. There exist two linearizations. One of them forces $p \succeq r$ and $q \succeq r$, the other forces $r \succeq p$ and $r \succeq q$. The two equivalence classes $\{p, q\}$ and $\{r\}$ are preserved (i.e., the only new relationships are those listed above).

By analogy with the dependency relation, we write $A \succ B$ if $A \succeq B$ and $B \not\succeq A$, and denote by $\max_{\succeq}(S)$ the set of all $A \in S$ such that for no $B \in S$, $B \succ A$. Note that as a consequence of the linearization process, if A and B belong to $\max_{\succeq}(S)$, then $A \sim B$.

Proposition 2. *For all disjunctive programs P, the corresponding dependency relation $\geq$ admits a linearization.*

We are ready to relax shift consistency.

Definition 5. *Let $\succeq$ be a linearization of $\geq$. We say that P is* weakly shift consistent w.r.t. $\succeq$ *if the following conditions hold:*

1. *P is order consistent.*
2. *For all $R \in Ground(P)$ and all distinct A and B in $\max_{\succeq}(head(R))$, $A \geq_{-} B$.*

We say that P is weakly shift consistent if there exists a linearization $\succeq$ of $\geq$ such that P is weakly shift consistent w.r.t. $\succeq$.

Next we show that the weakened definition is actually implied by "plain" shift consistency.

Proposition 3. *If P is shift consistent then P is weakly shift consistent, w.r.t. all linearizations of $\geq$.*

On the other hand, there exist weakly shift consistent programs that are not shift consistent.

Example 5. Let P consist of the rules:

$$R_1 = a \vee b \vee c$$
$$R_2 = b \leftarrow c$$
$$R_3 = c \leftarrow b.$$

Here there exists an equivalence class $\{b, c\}$ in $\max_{\geq}(head(R_1))/\sim$, such that $b \geq_{+} c$, therefore P is not shift consistent because Definition 3.(2) is violated. On the other hand, by choosing $\succeq$ such that $a \succeq b$ and $a \succeq c$, we obtain $\max_{\succeq}(head(R_1)) = \{a\}$, and hence Definition 5.(2) is satisfied. Indeed, P is weakly shift consistent, and has the following order-consistent normal shift:

$$R_1' = a \leftarrow \neg b, \neg c$$
$$R_2 = b \leftarrow c$$
$$R_3 = c \leftarrow b.$$

The generalized results depending on the relaxed shift consistency condition are the following.

Lemma 3. *All finitely recursive and weakly shift consistent disjunctive programs P have an order consistent normal shift P^s.*

Proof. (Sketch) Consider the dependency relation $\geq$ associated to the dependency graph of P, $DG(P)$. Let $\succeq$ be a linearization of $\geq$ satisfying the two conditions of Definition 5. Select one atom $A_R \in \max_{\succeq}(head(R))$ for each $R \in P$, and obtain P^s by shifting all the atoms in $head(R)\backslash\{A_R\}$ to the body (for all $R \in P$). Now suppose that the dependency graph $DG(P^s)$ contains an infinite chain $\mathcal{C} = A_0 \geq_{\pm} A_1 \geq_{\pm} \ldots \geq_{\pm} A_i \geq_{\pm} \ldots$ Since shifts introduce only negative literals, there must be a corresponding positive chain $A_0 \geq_{+}^{P} A_1 \geq_{+}^{P} \ldots \geq_{+}^{P} A_i \geq_{+}^{P} \ldots$ that must contain finitely many atoms, because P is finitely recursive by assumption. Then we may assume without loss of generality that all the atoms in $\mathcal{C}$ belong to the same strongly connected component of $DG(P)$, i.e., $A_i \sim^{P} A_j$ for all $i, j \geq 0$. This fact and Definition 5.(2) imply that the shifts applied to P do not introduce any new negative edges from A_i to A_j. It follows that $\mathcal{C}$ must also be a chain in $DG(P)$, but then Definition 5.(1) would be violated. We conclude that $\mathcal{C}$ cannot exist, that is, P^s is order consistent. □

Now the strengthened consistency theorem follows immediately from Lemma 3 and Lemma 1.

Theorem 5. *All finitely recursive and weakly shift consistent disjunctive programs P have at least one stable model.*

The relaxed shift consistency conditions impose weaker restrictions on rule heads. In particular, point 1 of the next proposition is less restrictive than the corresponding point in Proposition 1.

Proposition 4. *Let P be weakly shift consistent w.r.t. $\succeq$, and let R be any rule in P.*

1. $|\max_{\succeq}(head(R))| \leq 2$.
2. *If* $\max_{\succeq}(head(R)) = \{A, B\}$ *and* $A \neq B$, *then* $A \not\trianglerighteq B$ *and* $B \not\trianglerighteq A$.

Finding a linearization that makes P weakly shift consistent may be a complex process. An exact characterization of the computational complexity of this problem is left for further work.

We conclude this section by illustrating why P is assumed to be finitely recursive in the consistency theorems. If not, an order consistent normal shift might not exist even if P is shift consistent.

Example 6. Let P consist of the rules:

$$R_1 = p(X) \vee p(s(X))$$
$$R_2 = p(X) \leftarrow p(s(X))$$
$$R_3 = p(0) \leftarrow p(0)\,.$$

P is shift consistent, because (i) P is positive, and (ii) $\max_{\geq}(head(R_2\theta)) = \{p(X)\theta\}$, for all $R\theta \in Ground(P)$, and hence the two conditions of Definition 3 are trivially satisfied. However, P is not finitely recursive, because of R_2. There are only two possible normal shifts:

1. R_1 is transformed into $p(X) \leftarrow \neg p(s(X))$. This shift is not order consistent because of the infinite chain $p(0) \geq_{\pm} p(s(0)) \geq_{\pm} p(s(s(0))) \ldots$
2. R_1 is transformed into $p(s(X)) \leftarrow \neg p(X)$. In this case the shift is not order consistent because for all ground terms t there exists an infinite chain $p(t) \geq_{\pm} p(t) \geq_{\pm} p(t) \ldots$

Both shifts have no stable models.

4 Disjunctive Finitary Programs

In this section we apply the consistency results proved in Section 3 to extend the finitary program framework of [1] to disjunctive programs. First we need some terminology.

Definition 6. *Let F be a ground formula. A ground atom A is called a* kernel F-relevant atom *(w.r.t. a disjunctive program P) if A satisfies some of the following conditions:*

1. *A occurs in F.*
2. *There exists an infinite sequence $A \geq_{\pm} B_1 \geq_{\pm} B_2 \ldots \geq_{\pm} B_i \ldots$*
3. *For some $R \in P$, $A \in \max_{\geq}(head(R))$ and there exists $B \in \max_{\geq}(head(R))$ such that $A \not= B$, $A \sim B$, and $A \geq_{+} B$.*[2]

Next we define the *relevant universe and program* of a disjunctive program.

Definition 7. *The* relevant universe *for a ground formula F (w.r.t. program P), denoted by $U(P, F)$, is the set of all ground atoms A such that for some kernel F-relevant atom B, either $B \geq A$ or $\{A, B\} \subseteq head(R)$, for some $R \in Ground(P)$.*

The relevant subprogram *for a ground formula F (w.r.t program P), denoted by $R(P, F)$, is the set of all rules in $Ground(P)$ whose head belongs to $U(P, F)$.*

The relevant subprogram $R(P, F)$ suffices to answer queries about F.

Lemma 4. *For all ground formulae F, $R(P, F)$ has a stable model M_F iff P has a stable model M such that $M \cap U(P, F) = M_F$.*

Proof. (Sketch)

("If" part) Suppose M is a stable model of P. It can be verified that $U(P, F)$ is a splitting set for P and $R(P, F) = b_{U(P,F)}(P)$. Then, by the Splitting Theorem, there exist a stable model I of $R(P, F)$, and a stable model J of $e_{U(P,F)}(P \setminus R(P, F), I)$, such that $M = I \cup J$. By definition, no atom in $U(P, F)$ occurs in $e_{U(P,F)}(P \setminus R(P, F), I)$, therefore $J \cap U(P, F) = \emptyset$. It follows that $M \cap U(P, F) = I$, and the "If" part follows with $M_F = I$.

("Only if" part) Suppose $R(P, F)$ has a stable model M_F. By definition, all the ground atoms occurring in some infinite chain $A_1 \geq_{\pm} A_2 \geq_{\pm} \ldots \geq_{\pm}$

[2] Note that R violates Proposition 1, and hence the subprogram on which A depends is not shift consistent.

$A_i \geq_{\pm} \ldots$ belong to $U(P,F)$. Consequently, the dependency graph of $e_{U(P,F)}(P \setminus R(P,F), I)$ contains no such chains, i.e.
$e_{U(P,F)}(P \setminus R(P,F), I)$ is order consistent. Then, by Theorem 1, $e_{U(P,F)}(P \setminus R(P,F), I)$ has a stable model J. Let $M = J \cup M_F$. By the splitting theorem, M is a stable model of P. Moreover, since $J \cap U(P,F) = \emptyset$ (cf. "If" part), $M \cap U(P,F) = M_F$. □

Theorem 6. *For all ground formulae F,*

1. *P credulously entails F iff $R(P,F)$ credulously entails F.*
2. *P skeptically entails F iff $R(P,F)$ skeptically entails F.*

Proof. if P credulously entails F, then there exists a stable model M of P such that $M \models F$. By Lemma 4, $M \cap U(P,F)$ is a stable model of $R(P,F)$. Moreover, since by definition $U(P,F)$ contains all the atoms occurring in F, F must have the same truth value in M and $M \cap U(P,F)$, and hence $M \cap U(P,F) \models F$. As a consequence $R(P,F)$ credulously entails F.

Conversely, suppose that $R(P,F)$ credulously entails F. Then there exists a stable model M_F of $R(P,F)$ such that $M_F \models F$. By Lemma 4, P has a stable model M such that $M \cap U(P,F) = M_F$. Then the models M and M_F must agree on the valuation of F (cf. the "only if" part of the proof) and hence $M \models F$, which means that P credulously entails F. This completes the proof of 1).

To prove 2), we demonstrate the equivalent statement:

P does not skeptically entail F iff $R(P,F)$ does not skeptically entail F.

This statement is equivalent to: P credulously entails $\neg F$ iff $R(P,F)$ credulously entails $\neg F$, that follows immediately from 1). □

With this theorem, the compactness and completeness results of [1] will be extended to disjunctive programs.

First we extend the class of finitary normal programs to disjunctive programs as follows.

Definition 8. *A disjunctive program P is* finitary *if it satisfies the following conditions:*

1. *P is finitely recursive*
2. *there are finitely many odd-cyclic ground atoms*
3. *finitely many ground atoms satisfy condition 3 of Definition 6.*

A very simple example of disjunctive finitary program is illustrated in Figure 2. Condition 1 is guaranteed by the fact that for each rule, all the nonground terms in the body occur in the head, too. Condition 2 is trivially satisfied as there are no odd-cycles. Condition 3 is trivially satisfied as there is no positive dependency between `s(A)` and `ns(A)`, for all terms `A`.

Figure 3 illustrates a more complex program (using the machine syntax of DLV) that models the search space of a problem in reasoning about action and change. Here condition 1 is guaranteed because the time argument never

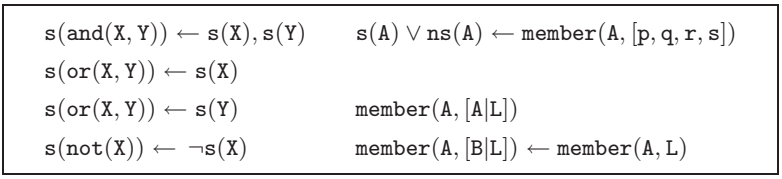


Fig. 2. A finitary disjunctive program for SAT

increases during recursive calls. Moreover, there are no odd-cycles, nor positive dependencies between the atoms of disjunctive heads.

Both examples are accepted by the finitary program recognizer demonstrated at LPNMR'01 and described in [2].

Proposition 5. *If a disjunctive program P is finitary then, for all ground goals G, $U(P,G)$ and $R(P,G)$ are finite.*

From this proposition we obtain the following results, that extend the major properties of normal finitary programs to disjunctive programs. The compactness theorem needs the following definition.

Definition 9 ([1]). *An* unstable kernel *for a program P is a subset K of $Ground(P)$ with the following properties:*

1. *K is* downward closed, *that is, for each atom A occurring in K, K contains all the rules $r \in Ground(P)$ with $A \in head(R)$.*
2. *K has no stable models.*

Theorem 7 (Compactness). *A finitary disjunctive program P has no stable models iff it has a finite unstable kernel.*

Proof. Let G be any ground atom in the language of P. From Lemma 4, it follows that P has no stable models iff $R(P,G)$ has no stable models. Clearly, $R(P,G)$ is downward closed by definition. Moreover, by Proposition 5, $R(P,G)$ is finite. Therefore P has no stable models iff $R(P,G)$ is a finite unstable kernel of P. □

Theorem 8. *For all finitary disjunctive programs P and ground goals G, both the problem of deciding whether G is a credulous consequence of P and the problem of deciding whether G is a skeptical consequence of P are decidable.*

Proof. By Theorem 6, G is a credulous (resp. skeptical) consequence of P iff G is a credulous (resp. skeptical) consequence of $R(P,G)$. Moreover, by Proposition 5, $R(P,G)$ is finite, so the set of its stable models can be computed in finite time. It follows that the inference problems for P and G are both decidable. □

Note that the above proof suggests how to implement finitary programs. It suffices to build a front-end that computes $R(P,G)$ (which is a finite ground program) and feed the result to one of the existing engines for answer set programming.

```
/* Frame axiom */

holds(P,T+1) :- holds(P,T), not ab(P,T).

/* Sample deterministic action */

holds( on_top(A,B), T+1) :-
        do( put_on(A,B), T),          /* action   */
        holds( is_clear(B), T),       /* preconds */
        holds( in_hand(A), T).

ab( on_top(A,C), T ) :-
        block(B),
        do( put_on(A,B), T),          /* action   */
        holds( is_clear(B), T),       /* preconds */
        holds( in_hand(A), T).


/* Sample nondeterministic action */

holds( in_hand(B), T+1) :-
        do( grasp(B), T),             /* action   */
        holds( is_clear(B), T),       /* preconds */
        not fails( grasp(B), T).

holds( on_table(B), T+1) :-
        do( grasp(B), T),             /* action   */
        holds( is_clear(B), T),       /* preconds */
        fails( grasp(B), T).

ab( on_top(B,C), T) :-
        do( grasp(B), T),             /* action   */
        holds( is_clear(B), T).       /* preconds */

fails( grasp(B), T) V succeeds( grasp(B), T) :- do( grasp(B), T).

/* Generate plan search space */

do( Act, T) V other_act( Act, T) :- action(Act).
```

Fig. 3. A finitary disjunctive program for reasoning about action and change

Theorem 9. *For all finitary disjunctive programs P and all goals G, both the problem of deciding whether $\exists G$ is a credulous consequence of P and the problem of deciding whether $\exists G$ is a skeptical consequence of P are semi-decidable.*

Proof. The formula $\exists G$ is credulously (res. skeptically) entailed by P iff there exists a grounding substitution θ such that $G\theta$ is credulously (res. skeptically) en-

tailed by P. The latter problem is decidable (by Theorem 8), and all grounding θ for G can be recursively enumerated, so existential entailment can be reduced to a potentially infinite recursive sequence of decidable tests, that terminates if and only if some $G\theta$ is entailed. □

Moreover, all the undecidability and Turing completeness results of [1] (i.e., all the lower bounds on inference complexity) can be immediately extended to disjunctive finitary programs because this class of programs includes normal finitary programs as a special case. Here is a brief summary of undecidability results:

- Finitary programs can simulate arbitrary Turing machines. More precisely, for each Turing machine $\mathcal{M}$ with initial state s and tape τ a (positive) finitary program P and a goal $\mathtt{p}(\mathtt{L},\mathtt{R},\mathtt{X})$ can be recursively constructed, in such a way that for all grounding substitutions θ, $P \models \mathtt{p}(\mathtt{L},\mathtt{R},\mathtt{X})\theta$ iff $\mathcal{M}$ terminates and $\mathtt{X}\theta$ encodes the final tape of the computation.
- As a consequence, credulous and skeptical nonground goals are strictly semi-decidable.
- For the class of all programs satisfying conditions 2 and 3 of Definition 8, credulous and skeptical inference are not semidecidable.
- For the class of all programs satisfying conditions 1 and 3 of Definition 8, credulous and skeptical inference are not semidecidable.

The last two points show that conditions 1 and 2 of Definition 8 are in some sense necessary for computability. Condition 3 will be discussed in Section 5.

4.1 A Note on Normal Programs

Definitions 6 and 7 correct an error in [1]. If we adapted the definitions in [1] to the terminology adopted in this paper, then Definition 6.(2) would be simply: "A occurs in an odd-cycle". Unfortunately, this is not enough to make Lemma 4 valid.

Example 7. Let $P = \{q(0),\ p(X) \leftarrow p(s(X)),\ p(X) \leftarrow \neg p(s(X))\}$. This program has no odd-cycles, so the relevant subprogram $R(P, q(0))$ equals $\{q(0)\}$ under the old definition. Now $R(P, q(0))$ has a stable model $M_F = \{q(0)\}$ while P has no stable model, therefore Lemma 4 is not valid under the old definitions.

It should be pointed out that all the results on finitary normal programs stated in [1] (including those proved by means of the old version of Lemma 4) are correct, because finitary programs are finitely recursive, and for these programs Definition 6.(2) is in fact equivalent to: "A occurs in an odd-cycle".

Summarizing, a correct definition of relevant universe for *normal* programs (that makes Lemma 4 valid) can be obtained by specializing Definition 7 as follows:

> $U(P, F)$ is the set of all ground atoms A such that there exists $B \geq A$, where either B occurs in F or there exists an infinite sequence $B \geq_{\pm} B_1 \geq_{\pm} B_2 \ldots \geq_{\pm} B_i \ldots$

5 Discussion and Perspectives

We proved that all the properties of normal finitary programs can be extended to disjunctive finitary programs. To do so, we generalized Fages theorem to a large class of disjunctive programs. Moreover, we have fixed an error in [1] that invalidated Lemma 4.

In some sense, however, one property of normal finitary programs has not been completely extended to the disjunctive case: currently, we cannot prove that Definition 8 is minimal, i.e., that *all* the 3 conditions defining disjunctive finitary programs are necessary to prove their properties. If we dropped any of the first two conditions, then inference would not be semidecidable anymore (the results for finitary normal programs immediately apply, see the discussion after Theorem 9). However, proving that the third condition is necessary amounts to refute the conjecture formulated in Section 3 (it states that all order consistent and finitely recursive disjunctive programs have a stable model). Therefore, the minimality of Definition 8 is still an open issue.

Furthermore, it remains to be seen how to exploit the weak form of shift consistency (Definition 5) for extending the class of disjunctive finitary programs. In practice, the problem is finding "good" linearizations that satisfy the analogous of the third condition of Definition 8. It is important to understand the computational complexity of this problem.

References

1. P. A. Bonatti. Reasoning with infinite stable models. *Proc. of IJCAI'01*, pp. 603-608, Morgan Kaufmann, 2001. 333, 334, 336, 341, 342, 343, 345, 346
2. P. A. Bonatti. Prototypes for reasoning with infinite stable models and function symbols. *Proc. of LPNMR'01*, pp. 416-419, LNAI 2173, Springer, 2001. 334, 343
3. P. A. Bonatti. Shift-based semantics: general results and applications. Technical report CD-TR 93/59, Technical University of Vienna, Computer Science Department, Institute of Information Systems, 1993. 335
4. P. Cholewiński, V. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with nonmonotonic reasoning. In *Proc. of ICLP'95*. MIT Press, 1995. 336
5. J. Dix, U. Furbach, A. Nerode. *Logic Programming and Nonmonotonic Reasoning: 4th international conference, LPNMR'97*, LNAI 1265, Springer Verlag, Berlin, 1997. 346
6. T. Eiter and G. Gottlob. Complexity results for disjunctive logic programming and applications to nonmonotonic logics. In *Proc. of ILPS'93*. MIT Press, 1993. 336
7. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello. A deductive system for nonmonotonic reasoning. In [5]. 333, 334
8. F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science* 1:51-60, 1994. 335
9. G. Gottlob. Complexity results for nonmonotonic logics. *Journal of Logic and Computation*, 2:397–425, 1992. 336
10. V. Lifschitz, H. Turner. Splitting a logic program. In *Proc. ICLP'94*, pp.23-37, MIT Press, 1994. 335

11. T. Syrjänen. Omega-restricted logic programs. *Proc. of LPNMR'01*, LNAI 2173, pp. 267-280, Springer, 2001. 333

Computing Stable Models: Worst-Case Performance Estimates

Zbigniew Lonc[1] and Mirosław Truszczyński[2]

[1] Faculty of Mathematics and Information Science, Warsaw University of Technology
00-661 Warsaw, Poland
[2] Department of Computer Science, University of Kentucky, Lexington
KY 40506-0046, USA

Abstract. We study algorithms for computing stable models of propositional logic programs and derive estimates on their worst-case performance that are asymptotically better than the trivial bound of $O(m2^n)$, where m is the size of an input program and n is the number of its atoms. For instance, for programs, whose clauses consist of at most two literals (counting the head) we design an algorithm to compute stable models that works in time $O(m \times 1.44225^n)$. We present similar results for several broader classes of programs, as well.

1 Introduction

The stable-model semantics was introduced by Gelfond and Lifschitz [GL88] to provide an interpretation for the negation operator in logic programming. In this paper, we study algorithms to compute stable models of propositional logic programs. Our goal is to design algorithms for which one can derive non-trivial worst-case performance bounds.

Computing stable models is important. It allows us to use logic programming with the stable-model semantics as a *computational* knowledge representation tool and as a declarative programming system. In most cases, when designing algorithms for computing stable models we restrict the syntax to that of DATALOG *with negation* (DATALOG$^\neg$), by eliminating function symbols from the language. When function symbols are allowed, models can be infinite and highly complex, and the general problem of existence of a stable model of a finite logic program is not even semi-decidable [MNR94]. However, when function symbols are not used, stable models are guaranteed to be finite and can be computed.

To compute stable models of *finite* DATALOG$^\neg$ programs we usually proceed in two steps. In the first step, we *ground* an input program P and produce a *finite* propositional program with the same stable models as P (finiteness of the resulting *ground* program is ensured by finiteness of P and absence of function symbols). In the second step, we compute stable models of the ground program by applying search. This general approach is used in *smodels* [NS00] and *dlv* [EFLP00], two most advanced systems to process DATALOG$^\neg$ programs.

It is this second step, computing stable models of propositional logic programs (in particular, programs obtained by grounding DATALOG$^\neg$ programs),

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 347–362, 2002.
© Springer-Verlag Berlin Heidelberg 2002

that is of interest to us in the present paper. Stable models of a propositional logic program P can be computed by a trivial brute-force algorithm that generates all subsets of the set of atoms of P and, for each of these subsets, checks the stability condition. This algorithm can be implemented to run in time $O(m2^n)$, where m is the size of P and n is the number of atoms in P (we will use m and n in this meaning throughout the paper). The algorithms used in *smodels* and *dlv* refine this brute-force algorithm by employing effective search-space pruning techniques. Experiments show that their performance is much better than that of the brute-force algorithm. However, at present, no non-trivial upper bound on their worst-case running time is known. In fact, *no* algorithms for computing stable models are known whose worst-case performance is *provably* better than that of the brute-force algorithm. Our main goal is to design such algorithms.

To this end, we propose a general template for an algorithm to compute stable models of propositional programs. The template involves an auxiliary procedure whose particular instantiation determines the specific algorithm and its running time. We propose concrete implementations of this procedure and show that the resulting algorithms for computing stable models are asymptotically better than the straightforward algorithm described above. The performance analysis of our algorithms is closely related to the question of how many stable models logic programs may have. We derive bounds on the maximum number of stable models in a program with n atoms and use them to establish lower and upper estimates on the performance of algorithms for computing all stable models.

Our main results concern propositional logic programs, called t-programs, in which the number of literals in rules, including the head, is bounded by a constant t. Despite their restricted syntax t-programs are of interest. Many logic programs that were proposed as encodings of problems in planning, model checking and combinatorics become propositional 2- or 3-programs after grounding. In general, programs obtained by grounding finite DATALOG$^\neg$ programs are t-programs, for some fixed, and usually small, t.

In the paper, for every $t \geq 2$, we construct an algorithm that computes all stable models of a t-program P in time $O(m\alpha_t^n)$, where α_t is a constant such that $\alpha_t < 2 - 1/2^t$. For 2-programs we obtain stronger results. We construct an algorithm that computes all stable models of a 2-program in time $O(m3^{n/3}) = O(m \times 1.44225^n)$. We note that $1.44225 < \alpha_2 \approx 1.61803$. Thus, this algorithm is indeed a significant improvement over the algorithm following from general considerations discussed above. We obtain similar results for a subclass of 2-programs consisting of programs that are purely negative and do not contain *dual* clauses. We also get significant improvements in the case when $t = 3$. Namely, we describe an algorithm that computes all stable models of a 3-program P in time $O(m \times 1.70711^n)$. In contrast, since $\alpha_3 \approx 1.83931$, the algorithm implied by the general considerations runs in time $O(m \times 1.83931^n)$.

In the paper we also consider a general case where no bounds on the length of a clause are imposed. We describe an algorithm to compute all stable models of such programs. Its worst-case complexity is slightly lower than that of the brute-force algorithm.

It is well known that, by introducing new atoms, every logic program P can be transformed in polynomial time into a 3-program P' that is, essentially, equivalent to P: every stable model of P is of the form $M' \cap At$, for some stable model M' of P' and, for every stable model M' of P', the set $M' \cap At$ is a stable model of P. This observation might suggest that in order to design fast algorithms to compute stable models, it is enough to focus on the class of 3-programs. It is not the case. In the worst case, the number of new atoms that need to be introduced is of the order of the size of the original program P. Consequently, an algorithm to compute stable models that can be obtained by combining the reduction described above with an algorithm to compute stable models of 3-programs runs in time $O(m2^m)$ and is asymptotically slower than the brute-force approach outlined earlier. Thus, it is necessary to study algorithms for computing stable models designed explicitly for particular classes of programs.

2 Preliminaries

For a detailed account of logic programming and stable model semantics we refer the reader to [GL88, Apt90, MT93]. In the paper, we consider only the propositional case. For a logic program P, by $At(P)$ we denote the set of all atoms appearing in P. We define $Lit(P) = At(P) \cup \{\mathbf{not}(a) : a \in At(P)\}$ and call elements of this set *literals*. Literals b and $\mathbf{not}(b)$, where b is an atom, are *dual* to each other. For a literal β, we denote its dual by $\mathbf{not}(\beta)$.

A *clause* is an expression c of the form $p \leftarrow B$ or $\leftarrow B$, where p is an atom and B is a set of literals (no literals in B are repeated). The clause of the first type is called *definite*. The clause of the second type is called a *constraint*. The atom p is the *head* of c and is denoted by $h(c)$. The set of atoms appearing in literals of B is called the *body* of c. The set of all positive literals (atoms) in B is the *positive body* of c, $b^+(c)$, in symbols. The set of atoms appearing in negated literals of B is the *negative body* of c, $b^-(c)$, in symbols.

A *logic program* is a collection of clauses. If every clause of P is definite, P is a *definite logic program*. If every clause in P has an empty positive body, that is, is *purely negative*, P is a *purely negative* program. Finally, a logic program P is a *t-program* if every clause in P has no more than t literals (counting the head).

A clause c is a *tautology* if it is definite and $h(c) \in b^+(c)$, or if $b^+(c) \cap b^-(c) \neq \emptyset$. A clause c is a *virtual constraint* if it is definite and $h(c) \in b^-(c)$. We have the following result [Dix95].

Proposition 1. *Let P be a logic program and let P' be the subprogram of P obtained by removing from P all tautologies, constraints and virtual constraints. If M is a stable model of P then it is a stable model of P'.*

Thanks to this proposition, when designing algorithms for computing stable models we may restrict attention to definite programs without tautologies and virtual constraints.

For a set of literals $L \subseteq Lit(P)$, we define:

$$L^+ = \{a \in At(P) : a \in L\} \quad \text{and} \quad L^- = \{a \in At(P) : \mathbf{not}(a) \in L\}.$$

We also define $L^0 = L^+ \cup L^-$. A set of literals L is *consistent* if $L^+ \cap L^- = \emptyset$. A set of atoms $M \subseteq At(P)$ is *consistent* with a set of literals $L \subseteq Lit(P)$, if $L^+ \subseteq M$ and $L^- \cap M = \emptyset$.

To characterize stable models of a program P that are consistent with a set of literals $L \subseteq Lit(P)$, we introduce a *simplification* of P with respect to L. By $[P]_L$ we denote the program obtained by removing from P

1. every clause c such that $b^+(c) \cap L^- \neq \emptyset$
2. every clause c such that $b^-(c) \cap L^+ \neq \emptyset$
3. every clause c such that $h(c) \in L^0$
4. every occurrence of a literal in L from the bodies of the remaining clauses.

The simplified program $[P]_L$ contains all information necessary to reconstruct stable models of P that are consistent with L. The following result was obtained in [Dix95] (we refer also to [SNV95, CT99]).

Proposition 2. *Let P be a logic program and L be a set of literals of P. If M is a stable model of P consistent with L, then $M \setminus L^+$ is a stable model of $[P]_L$.*

Thus, to compute all stable models of P that are consistent with L, one can first check if L is consistent. If not, there are no stable models consistent with L. Otherwise, one can compute all stable models of $[P]_L$, for each such model M' check whether $M = M' \cup L^+$ is a stable model of P and, if so, output M. This approach is the basis of the algorithm to compute stable models that we present in the following section.

3 A High-Level View of Stable Model Computation

We will now describe an algorithm *stable*(P, L) that, given a *definite* program P and a set of literals L, outputs all stable models of P that are consistent with L. The key concept we need is that of a complete collection. Let P be a logic program. A nonempty collection $\mathcal{A}$ of nonempty subsets of $Lit(P)$ is *complete* for P if every stable model of P is consistent with at least one set $A \in \mathcal{A}$. Clearly, the collection $\mathcal{A} = \{\{a\}, \{\mathbf{not}(a)\}\}$, where a is an atom of P, is an example of a complete collection for P. In the description given below, we assume that *complete*(P) is a procedure that, for a program P, computes a collection of sets of literals that is complete for P.

stable(P, L)
(0) **if** L is consistent **then**
(1) **if** $[P]_L = \emptyset$ **then**
(2) check whether L^+ is a stable model of P and, if so, output it
(3) **else**

(4) $\mathcal{A} := complete([P]_L)$;
(5) **for every** $A \in \mathcal{A}$ **do**
(6) $stable(P, L \cup A)$
(7) **end of** *stable*.

Proposition 3. *Let P be a definite finite propositional logic program. For every $L \subseteq Lit(P)$, $stable(P, L)$ returns all stable models of P consistent with L.*

Proof: We proceed by induction on $|At([P]_L)|$. To start, let us consider a call to $stable(P, L)$ in the case when $|At([P]_L)| = 0$ and let M be a set returned by $stable(P, L)$. It follows that L is consistent and that M is a stable model of P. Moreover, since $M = L^+$, M is consistent with L. Conversely, let M be a stable model of P that is consistent with L. By Proposition 2, $M \setminus L^+$ is a stable model of $[P]_L$. Since L is consistent (as M is consistent with L) and $[P]_L = \emptyset$, $M \setminus L^+ = \emptyset$. Since M is consistent with L, $M = L^+$. Thus, M is returned by $stable(P, L)$.

For the inductive step, let us consider a call to $stable(P, L)$, where $|At([P]_L)| > 0$. Let M be a set returned by this call. Then M is returned by a call to $stable(P, L \cup A)$, for some $A \in \mathcal{A}$, where $\mathcal{A}$ is a complete family for $[P]_L$. Since elements of a complete family are nonempty and consist of literals actually occurring in $[P]_L$, $|At([P]_{L \cup A})| < |At([P]_L)|$. By the induction hypothesis it follows that M is a stable model of P consistent with $L \cup A$ and, consequently, with L.

Let us now assume that M is a stable model of P consistent with L. Then, by Proposition 2, $M \setminus L^+$ is a stable model of $[P]_L$. Since $\mathcal{A}$ (computed in line (4)) is a complete collection for $[P]_L$, there is $A \in \mathcal{A}$ such that $M \setminus L^+$ is consistent with A. Since $A \cap L = \emptyset$ (as $A \subseteq At([P]_L)$), M is a stable model of P consistent with $L \cup A$. Since $|At([P]_{L \cup A})| < |At([P]_L)|$, by the induction hypothesis it follows that M is output during the recursive call to $stable(P, L \cup A)$. □

We will now study the performance of the algorithm *stable*. In our discussion we follow the notation used to describe it. Let P be a definite logic program and let $L \subseteq Lit(P)$. Let us consider the following recurrence relation:

$$s(P, L) = \begin{cases} 1 & \text{if } [P]_L = \emptyset \text{ or } L \text{ is not consistent} \\ \sum_{A \in \mathcal{A}} s(P, L \cup A) & \text{otherwise.} \end{cases}$$

As a corollary to Proposition 3 we obtain the following result.

Corollary 1. *Let P be a finite definite logic program and let $L \subseteq Lit(P)$. Then, P has at most $s(P, L)$ stable models consistent with L. In particular, P has at most $s(P, \emptyset)$ stable models.*

We will use the function $s(P, L)$ to estimate not only the number of stable models in definite logic programs but also the running time of the algorithm *stable*. Indeed, let us observe that the total number of times we make a call to the algorithm *stable* when executing $stable(P, L)$ (including the "top-level" call to $stable(P, L)$) is given by $s(P, L)$. We associate each execution of the instruction (i), where $0 \leq i \leq 5$, with the call in which the instruction is executed.

Consequently, each of these instructions is executed no more than $s(P, L)$ times during the execution of *stable*(P, L).

Let m be the size of a program P. There are linear-time algorithms to check whether a set of atoms is a stable model of a program P. Thus, we obtain the following result concerned with the performance of the algorithm *stable*.

Theorem 1. *If the procedure complete runs in time $O(t(m))$, where m is the size of an input program P, then executing the call stable(P, L), where $L \subseteq Lit(P)$, requires $O(s(P, L)(t(m) + m))$ steps in the worst case.*

The specific bound depends on the procedure *complete*, as it determines the recurrence for $s(P, L)$. It also depends on the implementation of the procedure *complete*, as the implementation determines the second factor in the running-time formula derived above.

Throughout the paper (except for Section 7, where a different approach is used), we specify algorithms to compute stable models by describing particular versions of the procedure *complete*. We obtain estimates on the running time of these algorithms by analyzing the recurrence for $s(P, L)$ implied by the procedure *complete*. As a byproduct to these considerations, we obtain bounds on the maximum number of stable models of a logic program with n atoms.

4 *t*-Programs

In this section we will instantiate the general algorithm to compute stable models to the case of t-programs, for $t \geq 2$. To this end, we will describe a procedure that, given a definite t-program P, returns a complete collection for P.

Let P be a definite t-program and let $x \leftarrow \beta_1, \ldots, \beta_k$, where β_i are literals and $k \leq t - 1$, be a clause in P. For every $i = 1, \ldots, k$, let us define

$$A_i = \{\mathbf{not}(x), \beta_1, \ldots, \beta_{i-1}, \mathbf{not}(\beta_i)\}$$

It is easy to see that the family $\mathcal{A} = \{\{x\}, A_1, \ldots, A_k\}$ is complete for P. We will assume that this complete collection is computed and returned by the procedure *complete*. Clearly, computing $\mathcal{A}$ can be implemented to run in time $O(m)$.

To analyze the resulting algorithm *stable*, we use our general results from the previous section. Let us define

$$c_n = \begin{cases} K_t & \text{if } 0 \leq n < t \\ c_{n-1} + \ldots + c_{n-t} & \text{otherwise,} \end{cases}$$

where K_t is the maximum possible value of $s(P, L)$ for a t-program P and a set of literals $L \subseteq Lit(P)$ such that $|At(P)| - |L| \leq t$. We will prove that if P is a t-program, $L \subseteq Lit(P)$, and $|At(P)| - |L| \leq n$, then $s(P, L) \leq c_n$. We proceed by induction on n. If $n < t$, then the assertion follows by the definition of K_t. So, let us assume that $n \geq t$. If L is not consistent or $[P]_L = \emptyset$, $s(P, L) = 1 \leq c_n$. Otherwise,

$$s(P, L) = \sum_{A \in \mathcal{A}} s(P, L \cup A) \leq c_{n-1} + c_{n-2} + \ldots + c_{n-t} = c_n.$$

The inequality follows by the induction hypothesis, the definition of $\mathcal{A}$, and the monotonicity of c_n. The last equality follows by the definition of c_n. Thus, the induction step is complete.

The characteristic equation of the recurrence c_n is $x^t = x^{t-1} + \ldots + x + 1$. Let α_t be the largest real root of this equation. One can show that for $t \geq 2$, $1 < \alpha_t < 2 - 1/2^t$. In particular, $\alpha_2 \approx 1.61803$, $\alpha_3 \approx 1.83931$, $\alpha_4 \approx 1.92757$ and $\alpha_5 \approx 1.96595$. The discussion in Section 3 implies the following two theorems.

Theorem 2. *Let t be an integer, $t \geq 2$. There is an algorithm to compute stable models of t-programs that runs in time $O(m\alpha_t^n)$, where n is the number of atoms and m is the size of the input program.*

Theorem 3. *Let t be an integer, $t \geq 2$. There is a constant C_t such that every t-program P has at most $C_t\alpha_t^n$ stable models, where $n = |At(P)|$.*

Since for every t, $\alpha_t < 2$, we indeed obtain an improvement over the straightforward approach. However, the scale of the improvement diminishes as t grows.

To establish lower bounds on the number of stable models and on the worst-case performance of algorithms to compute them, we define $P(n,t)$ to be a logic program such that $|At(P)| = n$ and P consists of all clauses of the form

$$x \leftarrow \mathbf{not}(b_1), \ldots, \mathbf{not}(b_t),$$

where $x \in At(P)$ and $\{b_1, \ldots, b_t\} \subseteq At(P) \setminus \{x\}$ are different atoms. It is easy to see that $P(n,t)$ is a $(t+1)$-program with n atoms and that stable models of $P(n,t)$ are precisely those subsets of $At(P)$ that have $n-t$ elements. Thus, $P(n,t)$ has exactly $\binom{n}{t}$ stable models.

Clearly, the program $P(2t-1, t-1)$ is a t-program over the set of $2t-1$ atoms. Moreover, it has $\binom{2t-1}{t}$ stable models. Let $kP(2t-1, t-1)$ be the logic program formed by the disjoint union of k copies of $P(2t-1, t-1)$ (sets of atoms of different copies of $P(2t-1, t-1)$ are disjoint). It is easy to see that $kP(2t-1, t-1)$ has $\binom{2t-1}{t}^k$ stable models. As an easy corollary of this observation we obtain the following result.

Theorem 4. *Let t be an integer, $t \geq 2$. There is a constant D_t such that for every n there is a t-program P with at least $D_t \times \binom{2t-1}{t}^{n/2t-1}$ stable models.*

This result implies that every algorithm for computing all stable models of a t-program in the worst-case requires $\Omega(\binom{2t-1}{t}^{n/2t-1})$ steps, as there are programs for which at least that many stable models need to be output. These lower bounds specialize to approximately $\Omega(1.44224^n)$, $\Omega(1.58489^n)$, $\Omega(1.6618^n)$ and $\Omega(1.71149^n)$, for $t = 2, 3, 4, 5$, respectively.

5 2-Programs

Stronger results can be derived for more restricted classes of programs. We will now study the case of 2-programs and prove the following two theorems.

Theorem 5. *There is an algorithm to compute stable models of 2-programs that runs in time* $O(m3^{n/3}) = O(m \times 1.44225^n)$*, where* n *is the number of atoms in* P *and* m *is the size of* P*.*

Theorem 6. *There is a constant* C *such that every 2-program* P *with* n *atoms, has at most* $C \times 3^{n/3}$ $(\approx C \times 1.44225^n)$ *stable models.*

By Proposition 1, to prove these theorems it suffices to limit attention to the case of definite programs not containing tautologies and virtual constraints. We will adopt this assumption and derive both theorems from general results presented in Section 3.

Let P be a definite 2-program. We say that an atom $b \in At(P)$ is a *neighbor* of an atom $a \in At(P)$ if P contains a clause containing both a and b (one of them as the head, the other one appearing positively or negatively in the body). By $n(a)$ we will denote the number of neighbors of an atom a. Since we assume that our programs contain neither tautologies nor virtual constraints, no atom a is its own neighbor.

We will now describe the procedure *complete*. The complete family returned by the call to $complete(P)$ depends on the program P. We list below several cases that cover all definite 2-programs without tautologies and virtual constraints. In each of these cases, we specify a complete collection to be returned by the procedure *complete*.

Case 1. There is an atom, say x, such that P contains a clause with the head x and with the empty body (in other words, x is a fact of P). We define $\mathcal{A} = \{\{x\}\}$. Clearly, every stable model of P contains x. Thus, $\mathcal{A}$ is complete.

Case 2. There is an atom, say x, that does not appear in the head of any clause in P. We define $\mathcal{A} = \{\{\mathbf{not}(x)\}\}$. It is well known that x does not belong to *any* stable model of P. Thus, $\mathcal{A}$ is complete for P.

Case 3. There are atoms x and y, $x \not= y$, such that $x \leftarrow y$ and at least one of $x \leftarrow \mathbf{not}(y)$ and $y \leftarrow \mathbf{not}(x)$ are in P. In this case, we set $\mathcal{A} = \{\{x\}\}$. Let M be a stable model of P. If $y \in M$, then $x \in M$ (due to the fact that the clause $x \leftarrow y$ is in P). Otherwise, $y \notin M$. Since M satisfies $x \leftarrow \mathbf{not}(y)$ or $y \leftarrow \mathbf{not}(x)$, it again follows that $x \in M$. Thus, $\mathcal{A}$ is complete.

Case 4. There are atoms x and y such that $x \leftarrow y$ and $y \leftarrow x$ are both in P. We define

$$\mathcal{A} = \{\{x, y\}, \{\mathbf{not}(x), \mathbf{not}(y)\}\}.$$

If M is a stable model of P then, clearly, $x \in M$ if and only if $y \in M$. It follows that either $\{x, y\} \subseteq M$ or $\{x, y\} \cap M = \emptyset$. Thus, $\mathcal{A}$ is complete for P. Moreover, since $x \not= y$ (P does not contain clauses of the form $w \leftarrow w$), each set in $\mathcal{A}$ has at least two elements.

Case 5. None of the Cases 1-4 holds and there is an atom, say x, with exactly one neighbor, y. Since P does not contain clauses of the form $w \leftarrow w$ and $w \leftarrow \mathbf{not}(w)$, we have $x \not= y$. Moreover, x must be the head of at least one clause (since we assume here that Case 2 does not hold).

Subcase 5a. P contains the clause $x \leftarrow y$. We define

$$\mathcal{A} = \{\{x, y\}, \{\mathbf{not}(x), \mathbf{not}(y)\}\}.$$

Let M be a stable model of P. If $y \in M$ then, clearly, $x \in M$. Since we assume that Case 3 does not hold, the clause $x \leftarrow y$ is the only clause in P with x as the head. Thus, if $y \notin M$, then we also have that $x \notin M$. Hence, $\mathcal{A}$ is complete.
Subcase 5b. P does not contain the clause $x \leftarrow y$. We define

$$\mathcal{A} = \{\{x, \mathbf{not}(y)\}, \{\mathbf{not}(x), y\}\}.$$

Let M be a stable model of P. Since x is the head of at least one clause in P, it follows that the clause $x \leftarrow \mathbf{not}(y)$ belongs to P. Thus, if $y \notin M$ then $x \in M$. If $y \in M$ then, since $x \leftarrow \mathbf{not}(y)$ is the only clause in P with x as the head, $x \notin M$. Hence, $\mathcal{A}$ is complete.
Case 6. None of the Cases 1-5 holds. Let $w \in At(P)$ be an atom. By $x_1, \ldots, x_p$ we denote all atoms x in P such that $w \leftarrow \mathbf{not}(x)$ or $x \leftarrow \mathbf{not}(w)$ is a clause in P. Similarly, by $y_1, \ldots, y_q$ we denote all atoms y in P such that $y \leftarrow w$ is a clause of P. Finally, by $z_1, \ldots, z_r$ we denote all atoms z of P such that $w \leftarrow z$ is a clause of P. By our earlier discussion it follows that the sets $\{x_1, \ldots, x_p\}$, $\{y_1, \ldots, y_q\}$ and $\{z_1, \ldots, z_r\}$, are pairwise disjoint and cover all neighbors of w. That is, the number of neighbors of w is given by $p + q + r$. Since we exclude Case 5 here, $p+q+r \geq 2$. Further, since w is the head of at least one edge (Case 2 does not hold), it follows that $p + r \geq 1$
Subcase 6a. For some atom w, $q \geq 1$ or $p + q + r \geq 3$. Then, we define

$$\mathcal{A} = \{\{w, y_1, \ldots, y_q\}, \{\mathbf{not}(w), x_1, \ldots, x_p, \mathbf{not}(z_1), \ldots, \mathbf{not}(z_r)\}\}.$$

It is easy to see that $\mathcal{A}$ is complete for P. Moreover, if $q \geq 1$ then, since $p+r \geq 1$, each of the two sets in $\mathcal{A}$ has at least two elements. If $p + q + r \geq 3$, then either each set in $\mathcal{A}$ has at least two elements, or one of them has one element and the other one at least four elements.
Subcase 6b. Every atom w has exactly two neighbors, and does not appear in the body of any Horn clause of P. It follows that all clauses in P are purely negative. Let w be an arbitrary atom in P. Let u and v be the two neighbors of w. The atoms u and v also have two neighbors each, one of them being w. Let u' and v' be the neighbors of u and v, respectively, that are different from w. We define

$$\mathcal{A} = \{\{\mathbf{not}(w), u, v\}, \{\mathbf{not}(u), w, u'\}, \{\mathbf{not}(v), w, v'\}\}.$$

Let M be a stable model of P. Let us assume that $w \notin M$. Since w and u are neighbors, there is a clause in P built of w and u. This clause is purely negative and it is satisfied by M. It follows that $u \in M$. A similar argument shows that $v \in M$, as well. If $w \in M$ then, since M is a stable model of P, there is a 2-clause C in P with the head w and with the body satisfied by M. Since P consists of purely negative clauses, and since u and v are the only neighbors of w, $C = w \leftarrow \mathbf{not}(u)$ or $C = w \leftarrow \mathbf{not}(v)$. Let us assume the former. It is clear that $u \notin M$ (since M satisfies the body of C). Let us recall that u' is a neighbor of u. Consequently, u and u' form a purely negative clause of P. This clause is satisfied by M. Thus, $u' \in M$ and M is consistent with $\{\mathbf{not}(u), w, u'\}$.

In the other case, when $C = w \leftarrow \mathbf{not}(v)$, a similar argument shows that M is consistent with $\{\mathbf{not}(v), w, v'\}$. Thus, every stable model of P is consistent with one of the three sets in $\mathcal{A}$. In other words, $\mathcal{A}$ is complete.

Clearly, given a 2-program P, deciding which of the cases described above holds for P can be implemented to run in linear time. Once that is done, the output collection can be constructed and returned in linear time, too.

This specification of the procedure *complete* yields a particular algorithm to compute stable models of definite 2-programs without tautologies and virtual constraints. To estimate its performance and obtain the bound on the number of stable models, we define

$$c_n = \begin{cases} K & \text{if } 0 \leq n < 4 \\ \max\{c_{n-1}, 2c_{n-2}, c_{n-1} + c_{n-4}, 3c_{n-3}\} & \text{otherwise,} \end{cases}$$

where K is the maximum possible value of $s(P, L)$, when P is a definite finite propositional logic program, $L \subseteq Lit(P)$ and $|At(P)| - |L| \leq 3$. It is easy to see that K is a constant that depends neither on P nor on L. We will prove that $s(P, L) \leq c_n$, where $n = |At(P)| - |L|$. If $n \leq 3$, then the assertion follows by the definition of K. So, let us assume that $n \geq 4$. If L is not consistent or $[P]_L = \emptyset$, $s(P, L) = 1 \leq c_n$. Otherwise,

$$s(P, L) = \sum_{A \in \mathcal{A}} s(P, L \cup A) \leq \max\{c_{n-1}, 2c_{n-2}, c_{n-1} + c_{n-4}, 3c_{n-3}\} = c_n.$$

The inequality follows by the induction hypothesis, the properties of the complete families returned by *complete* (the cardinalities of sets forming these complete families) and the monotonicity of c_n.

Using well-known properties of linear recurrence relations, it is easy to see that $c_n = O(3^{n/3}) = O(1.44225^n)$. Thus, Theorems 5 and 6 follow.

As concerns bounds on the number of stable models of a 2-program, a stronger (exact) result can be derived. Let

$$g_n = \begin{cases} 3^{n/3} & \text{if } n = 0 \pmod 3 \\ 4 \times 3^{(n-4)/3} & \text{if } n = 1 \pmod 3, \text{ and } n > 1 \\ 2 \times 3^{(n-2)/3} & \text{if } n = 2 \pmod 3 \\ 1 & \text{if } n = 1 \end{cases}$$

Exploiting connections between stable models of purely negative definite 2-programs and maximal independent sets in graphs, and using some classic results from graph theory [MM65] one can prove the following result.

Corollary 2. *Let P be a 2-program with n atoms. Then P has no more than g_n stable models.*

The bound of Corollary 2 cannot be improved as there are logic programs that achieve it. Let $P(p_1, \ldots, p_k)$, where for every i, $p_i \geq 2$, be a disjoint union of programs $P(p_1, 1), \ldots, P(p_k, 1)$ (we discussed these programs in Section 2). Each program $P(p_i, 1)$ has p_i stable models. Thus, the number of stable models

of $P(p_1, \ldots, p_k)$ is $p_1 p_2 \ldots p_k$. Let P be a logic program with $n \geq 2$ atoms and of the form $P(3, \ldots, 3)$, $P(2, 3, \ldots, 3)$ or $P(4, 3, \ldots, 3)$, depending on $n (\mathrm{mod}\ 3)$. It is easy to see that P has g_n stable models. In particular, it follows that our algorithm to compute all stable models of 2-programs is must execute at least $\Omega(3^{n/3})$ steps in the worst case.

Narrowing the class of programs leads to still better bounds and faster algorithms. We will discuss one specific subclass of the class of 2-programs here. Namely, we will consider definite purely negative 2-programs with no *dual* clauses (two clauses are called *dual* if they are of the form $a \leftarrow \mathbf{not}(b)$ and $b \leftarrow \mathbf{not}(a)$). We denote the class of these programs by $\mathcal{P}_2^n$. Using the same approach as in the case of arbitrary 2-programs, we can prove the following two theorems.

Theorem 7. *There is an algorithm to compute stable models of 2-programs in the class $\mathcal{P}_2^n$ that runs in time $O(m \times 1.23651^n)$, where n is the number of atoms and m is the size of an input program.*

Theorem 8. *There is a constant C such that every 2-program $P \in \mathcal{P}_2^n$ has at most $C \times 1.23651^n$ stable models.*

Theorem 8 gives an upper bound on the number of stable models of a program in the class $\mathcal{P}_2^n$. To establish a lower bound, we define S_6 to be a program over the set of atoms $a_1, \ldots, a_6$ and containing the rules (the arithmetic of indices is performed modulo 6): $a_{i+1} \leftarrow \mathbf{not}(a_i)$ and $a_{i+2} \leftarrow \mathbf{not}(a_i)$, $i = 0, 1, 2, 3, 4, 5$. The program S_6 has three stable models: $\{a_0, a_1, a_3, a_4\}$, $\{a_1, a_2, a_4, a_5\}$ and $\{a_2, a_3, a_5, a_0\}$.

Let P be the program consisting of k copies of S_6, with mutually disjoint sets of atoms. Clearly, P has 3^k stable models. Thus, there is a constant D such that for every $n \geq 1$ there is a program P with n atoms and with at least $D \times 3^{n/6}$ ($\approx D \times 1.20094^n$) stable models.

6 3-Programs

We will now present our results for the class of 3-programs. Using similar techniques as those presented in the previous section, we prove the following two theorems.

Theorem 9. *There is an algorithm to compute stable models of 3-programs that runs in time $O(m \times 1.70711^n)$, where m is the size of the input.*

Theorem 10. *There is a constant C such that every 3-program P has at most $C \times 1.70711^n$ stable models.*

The algorithm whose existence is claimed in Theorem 9 is obtained from the general template described in Section 3 by a proper instantiation of the procedure *complete* (in a similar way to that presented in detail in the previous section for the case of 2-programs).

The lower bound in this case follows from an observation made in Section 4 that there is a constant D_3 such that for every n there is a 3-program P such that P has at least $D_3 \times 1.58489^n$) stable models (cf. Theorem 4). Thus, every algorithm for computing all stable models of 3-programs must take at least $\Omega(1.58489^n)$ steps in the worst case.

7 The General Case

In this section we present an algorithm that computes all stable models of arbitrary propositional logic programs. It runs in time $O(m2^n/\sqrt{n})$ and so, provides an improvement over the trivial bound $O(m2^n)$. However, our approach is quite different from that used in the preceding sections. The key component of the algorithm is an auxiliary procedure *stable_aux*(P, π). Let P be a logic program and let $At(P) = \{x_1, x_2, \ldots, x_n\}$. Given P and a permutation π of $\{1, 2, \ldots, n\}$, the procedure *stable_aux*(P, π) looks for an index j, $1 \leq j \leq n$, such that the set $\{x_{\pi(j)}, \ldots, x_{\pi(n)}\}$ is a stable model of P. Since no stable model of P is a proper subset of another stable model of P, for any permutation π there is at most one such index j. If such j exists, the procedure outputs the set $\{x_{\pi(j)}, \ldots, x_{\pi(n)}\}$.

In the description of the algorithm *stable_aux*, we use the following notation. For every atom a, by $pos(a)$ we denote the list of all clauses which contain a (as a non-negated atom) in their bodies, and by $neg(a)$ a list of all clauses that contain $\mathbf{not}(a)$ in their bodies. Given a standard linked-list representation of logic programs, all these lists can be computed in time linear in m.

Further, for each clause C, we introduce counters $p(C)$ and $n(C)$. We initialize $p(C)$ to be the number of positive literals (atoms) in the body of C. Similarly, we initialize $n(C)$ to be the number of negative literals in the body of C. These counters are used to decide whether a clause belongs to the reduct of the program and whether it "fires" when computing the least model of the reduct.

stable_aux(P, π)
(1) $M = At(P)$;
(2) $Q :=$ set of clauses C such that $p(C) = n(C) = 0$;
(3) $lm := \emptyset$;
(4) **for** $j = 1$ **to** n **do**
(5) **while** $Q \neq \emptyset$ **do**
(6) $C_0 :=$ any clause in Q;
(7) mark C_0 as used and remove it from Q;
(8) **if** $h(C_0) \notin lm$ **then**
(9) $lm := lm \cup \{h(C_0)\}$;
(10) **for** $C \in pos(h(C_0))$ **do**
(11) $p(C) := p(C) - 1$;
(12) **if** $p(C) = 0$ & $n(C) = 0$ & C not used **then** add C to Q;
(13) **if** $lm = M$ **then** output M and stop;
(14) $M := M \setminus \{x_{\pi(j)}\}$;
(15) **for** $C \in neg(x_{\pi(j)})$ **do**
(16) $n(C) := n(C) - 1$;
(17) **if** $n(C) = 0$ & $p(C) = 0$ & C not used **then** add C to Q.

Let us define $M_j = \{x_{\pi(j)}, \ldots, x_{\pi(n)}\}$. Intuitively, the algorithm *stable_aux* works as follows. In the iteration j of the **for** loop it computes the least model of the reduct P^{M_j} (lines (5)-(12)). Then it tests whether $M_j = lm(P^{M_j})$ (line (13)). If so, it outputs M_j (it is a stable model of P) and terminates. Otherwise, it computes the reduct $P^{M_{j+1}}$. In fact the reduct is not explicitly computed. Rather, the number of negated literals in the body of each rule is updated to reflect the fact that we shift attention from the set M_j to the set M_{j+1} (lines (14)-(17)). The key to the algorithm is the fact that it computes reducts P^{M_j} and least models $lm(P^{M_j})$ in an incremental way and, so, tests n candidates M_j for stability in time $O(m)$ (where m is the size of the program).

Proposition 4. *Let P be a logic program and let $At(P) = \{x_1, \ldots, x_n\}$. For every permutation π of $\{1, \ldots, n\}$, if $M = \{x_{\pi(j)}, \ldots, x_{\pi(n)}\}$ then the procedure stable_aux(P, π) outputs M if and only if M is a stable model of P. Moreover, the procedure stable_aux runs in $O(m)$ steps, where m is the size of P.*

We will now describe how to use the procedure *stable_aux* in an algorithm to compute stable models of a logic program. A collection $\mathcal{S}$ of permutations of $\{1, 2, \ldots, n\}$ is *full* if every subset S of $\{1, 2, \ldots, n\}$ is a final segment (suffix) of a permutation in $\mathcal{S}$ or, more precisely, if for every subset S of $\{1, 2, \ldots, n\}$ there is a permutation $\pi \in \mathcal{S}$ such that $S = \{\pi(n - |S| + 1), \ldots, \pi(n)\}$.

If S_1 and S_2 are of the same cardinality then they cannot occur as suffixes of the same permutation. Since there are $\binom{n}{\lfloor n/2 \rfloor}$ subsets of $\{1, 2, \ldots, n\}$ of cardinality $\lfloor n/2 \rfloor$, every full family of permutations must contain at least $\binom{n}{\lfloor n/2 \rfloor}$ elements. An important property is that for every $n \geq 0$ there is a full family of permutations of cardinality $\binom{n}{\lfloor n/2 \rfloor}$. An algorithm to compute such a minimal full set of permutations, say $\mathcal{S}_{min}$, is described in [Knu98] (Vol. 3, pages 579 and 743-744). We refer to this algorithm as *perm*(n). The algorithm *perm*(n) enumerates all permutations in $\mathcal{S}_{min}$ by generating each next permutation entirely on the basis of the previous one. The algorithm *perm*(n) takes $O(n)$ steps to generate a permutation and each permutation is generated only once.

We modify the algorithm *perm*(n) to obtain an algorithm to compute all stable models of a logic program P. Namely, each time a new permutation, say π, is generated, we make a call to *stable_aux*(P, π). We call this algorithm *stable*^{p}. Since $\binom{n}{\lfloor n/2 \rfloor} = \Theta(2^n/\sqrt{n})$ we have the following result.

Proposition 5. *The algorithm stable^p is correct and runs in time $O(m2^n/\sqrt{n})$.*

Since the program $P(n, \lfloor n/2 \rfloor)$ has exactly $\binom{n}{\lfloor n/2 \rfloor}$ stable models, every algorithm to compute all stable models of a logic program must take at least $\Omega(2^n/\sqrt{n})$ steps.

8 Discussion and Conclusions

We presented algorithms for computing stable models of logic programs with worst-case performance bounds asymptotically better than the trivial bound of

$O(m2^n)$. These are first results of that type in the literature. In the general case, we proposed an algorithm that runs in time $O(m2^n/\sqrt{n})$ improving the performance over the brute-force approach by the factor of $\sqrt{n}$. Most of our work, however, was concerned with algorithms for computing stable models of t-programs. We proposed an algorithm that computes stable models of t-programs in time $O(m\alpha_t^n)$, where $\alpha_t < 2 - 1/2^t$. We strengthened these results in the case of 2- and 3-programs. In the first case, we presented an algorithm that runs in time $O(m3^{n/3})$ ($\approx O(m \times 1.44225^n)$). For the case of 3-programs, we presented an algorithm running in the worst case in time $O(m \times 1.70711^n)$.

In addition to these contributions, our work leads to several interesting questions. A foremost among them is whether our results can be further improved. First, we observe that in the case when the task is to compute *all* stable models, we already have proved optimality (up to a polynomial factor) of the algorithms developed for the class of all programs and the class of all 2-programs. However, in all other cases there is still room for improvement — our lower and upper bounds do not coincide.

The situation gets even more interesting when we want to compute one stable model (if stable models exist) rather than all of them. Algorithms we presented here can, of course, be adapted to this case (by terminating them as soon as the first model is found). Thus, the upper bounds derived in this paper remain valid. But the lower bounds, which we derive on the basis of the number of stable models input programs may have, do not. In particular, it is no longer clear whether the algorithm we developed for the case of 2-programs remains optimal. One cannot exclude existence of pruning techniques that, in the case when the input program has stable models, would on occasion eliminate from considerations parts of the search space possibly containing some stable models, recognizing that the remaining portion of the search space still contains some.

Such search space pruning techniques are possible in the case of satisfiability testing. For instance, the pure literal rule, sometimes used by implementations of the Davis-Putnam procedure, eliminates from considerations parts of search space that may contain stable models [MS85, Kul99]. However, the part that remains is guaranteed to contain a model as long as the input theory has one. No examples of analogous search space pruning methods are known in the case of stable model computation. We feel that nonmonotonicity of the stable model semantics is the reason for that but a formal account of this issue remains an open problem.

Finally, we note that many algorithms to compute stable models can be cast as instantiations of the general template introduced in Section 3. For instance, it is the case with the algorithm used in *smodels*. To view *smodels* in this way, we define the procedure complete as (1) picking (based on full lookahead) an atom x on which the search will split; (2) computing the set of literals $A(x)$ by assuming that x holds and by applying the unit propagation procedure of *smodels* (based, we recall on the ideas behind the well-founded semantics); (3) computing in the same way the set $A(\mathbf{not}(x))$ by assuming that $\mathbf{not}(x)$ holds; and (4) returning the family $\mathcal{A} = \{A(x), A(\mathbf{not}(x))\}$. This family is clearly complete.

While different in some implementation details, the algorithm obtained from our general template by using this particular version of the procedure complete is essentially equivalent to that of *smodels*. By modifying our analysis in Section 5, one can show that on 2-programs *smodels* runs in time $O(m \times 1.46558^n)$ and on purely negative programs without dual clauses in time $O(m \times 1.32472^n)$. To the best of our knowledge these are first non-trivial estimates of the worst-case performance of *smodels*. These bounds are worse from those obtained from the algorithms we proposed here, as the techniques we developed were not designed with the analysis of *smodels* in mind. However, they demonstrate that the worst-case analysis of algorithms such as *smodels*, which is an important open problem, may be possible.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0097278.

References

[Apt90] K. Apt. Logic programming. In J. van Leeuven, editor, *Handbook of theoretical computer science*, pages 493–574. Elsevier, Amsterdam, 1990. 349

[BE96] P. A. Bonatti and T. Eiter. Querying Disjunctive Databases Through Nonmonotonic Logics. *Theoretical Computer Science*, 160:321–363, 1996.

[CT99] P. Cholewiński and M. Truszczyński. Extremal problems in logic programming and stable model computation. *Journal of Logic Programming*, 38:219–242, 1999. 350

[Dix95] J. Dix. A classification theory of semantics of normal logic programs: Ii. weak properties. *Fundamenta Informaticae*, 22(3):257 – 288, 1995. 349, 350

[EFLP00] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem-solving in DLV. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, Dordrecht, 2000. 347

[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. 347, 349

[Knu98] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley, 1998. Second edition. 359

[Kul99] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, pages 1–72, 1999. 360

[MM65] J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Math*, pages 23–28, 1965. 356

[MNR94] W. Marek, A. Nerode, and J. B. Remmel. The stable models of predicate logic programs. *Journal of Logic Programming*, 21(3):129–154, 1994. 347

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, pages 287–295, 1985. 360

[MT93] W. Marek and M. Truszczyński. *Nonmonotonic logics; context-dependent reasoning*. Springer-Verlag, Berlin, 1993. 349

[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. 347

[SNV95] V. S. Subrahmanian, D. Nau, and C. Vago. WFS + branch bound = stable models. *IEEE Transactions on Knowledge and Data Engineering*, 7:362–377, 1995. 350

Towards Local Search for Answer Sets

Yannis Dimopoulos and Andreas Sideris

Department of Computer Science, University of Cyprus
P.O. Box 20537, CY1678, Nicosia, Cyprus
yannis@cs.ucy.ac.cy
sideris@cytanet.com.cy

Abstract. Answer set programming has emerged as a new important paradigm for declarative problem solving. It relies on algorithms that compute the stable models of a logic program, a problem that is, in the worst-case, intractable. Although, local search procedures have been successfully applied to a variety of hard computational problems, the idea of employing such procedures in answer set programming has received very limited attention.
This paper presents several local search algorithms for computing the stable models of a normal logic program. They are all based on the notion of a *conflict set*, but use it in different ways, resulting in different computational behaviors. The algorithms are inspired from related work in solving propositional satisfiability problems, suitably adapted to the stable model semantics. The paper also discusses how the heuristic equivalence method, that has been proposed in the context of propositional satisfiability, can be used in systematic search procedures that compute the stable models of logic programs.

1 Introduction

Answer set programming [7] has been proposed as a new declarative logic programming approach that differs from the classical Prolog goal-directed backward chaining paradigm. In answer set programming, a problem is represented by a logic program whose *stable models* [6] correspond to the solutions of the problem. The success of answer set programming relies heavily on the derivation of effective algorithms for computing the stable models of logic programs. In recent years there has been remarkable progress in the development of systems that compute stable models, eg. `DLV` [2] and `Smodels` [22]. These systems have been applied to various problems such as planning [3], diagnosis [4] and model checking [9].

Almost all existing stable model algorithms are systematic procedures that explore the search space of a problem through the standard backtracking mechanism. Although, these procedures can be very effective in many problems, they may fail in cases where they end up in regions deep in the search space that do not contain any solution. Recently, [8] proposed a new method, called the heuristic equivalence method, that introduces randomization to systematic search procedures that solve propositional satisfiability problems. We implemented this

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 363–377, 2002.
© Springer-Verlag Berlin Heidelberg 2002

method in the Smodels system, and in this paper we report on some first experimental results.

An alternative to systematic search are the local search methods. These methods have been successfully applied to a variety of hard computational problems, including the problem of finding a satisfying truth assignment for a CNF formula (SAT). Despite this success, there has been no attempt to apply local search to the problem of computing the stable models of a logic program. The only related work, that is described in [12], translates disjunctive logic programs in CNF and then uses a local search SAT algorithm. However, in order to prove the minimality of the generated models it uses a systematic SAT procedure, and therefore can been seen as a combination of local and systematic search methods. Also related is the work described in [15], [16] and [1], where genetic algorithms and ant colony optimization techniques are applied to the problems of computing the extensions of default theories and the stable models of logic programs.

This paper is a first attempt towards introducing local search algorithms that are similar to SAT local search procedures, into answer set programming. We present several different algorithms that start with a random assignment on the atoms of a normal logic program and at each step change the value, or "flip", one of these atoms. The objective function that is minimized is the cardinality of what is called the *conflict set* of the program wrt to an assignment. A stable model is an assignment with an empty conflict set. The algorithms differ in the way they select the atoms that are candidates for flipping, as well as, in the heuristics they employ for actually flipping one of the candidates. Of course, local search procedures can not prove that a logic program does not have a stable model.

The local search algorithms are implemented in LSM , an experimental system that is currently under development. At its current first stage LSM serves the purpose of providing an environment for experimentation with different techniques that have been applied to solving SAT problems, suitably adapted to the peculiarities of stable models. LSM is implemented as an extension to Smodels and uses its syntax and much of its code.

2 Preliminaries

In this section we review the stable model semantics and some basic local search algorithms for the SAT problem.

2.1 The Stable Models Semantics

A *normal logic program* P is a set of *normal rules*. A normal rule r is a rule of the form

$$h \leftarrow a_1, a_2, ..., a_n, not\ b_1,\ not\ b_2, ...,\ not\ b_m$$

where $h, a_1, a_2, ..., a_n, b_1, b_2, ..., b_m$ are propositional atoms. We define $body(r)^- = \{b_1, b_2, ..., b_m\}$, $body(r)^+ = \{a_1, a_2, ..., a_n\}$, $body(r) = body(r)^- \cup body(r)^+$, and

$head(r) = h$. The set of atoms of P is denoted by $Atoms(P)$. An atom p prefixed with the operator *not* is called a *negated atom*. Atoms and negated atoms are both called literals.

Let S be a truth or value assignment on the atoms of $Atoms(P)$. We denote by S^+ the set of atoms of S that are assigned the value true, and S^- the set of atoms that are assigned the value false. The *reduct* P^S of a logic program P wrt to an assignment S is the logic program obtained from P after deleting

- every rule of P that has a negated atom *not* b_i, with $b_i \in S^+$
- every negated atom from the body of the remaining rules

The resulting program does not contain negated atoms, and is called a definite logic program. Let $cl(P)$ denote the deductive closure of a definite logic program P, which coincides with its minimal model. A *stable model* of a logic program P is an assignment S, such that $S = cl(P^S)$.

Recent versions of the `Smodels` system [20] extend the syntax of logic programs with a variety of rules that are more expressive than normal rules. In the following we discuss one particular type of such rules, that are called *choice rules*. Choice rules, which can be translated into a set of normal rules, are useful from a programming perspective, as they provide a concise way of representing knowledge.

In many problem domains, we are interested in computing a stable model that assigns true or false to specific atoms. In `Smodels` this is declared through the *compute* statement. For example, the statement $compute(\{q, not\ r\})$ in a program P, will generate stable models of P that assign true to q and false to r, or report failure if no such stable model exists. Although compute statements can be easily expressed by normal rules, atoms that appear in such statements have a fixed truth value in the stable models of interest, and therefore are treated in a slightly different way than other atoms of the program.

2.2 Local Search for Propositional Satisfiability

Local search algorithms have been applied with considerable success to the problem of propositional satisfiability (SAT), ie. the problem of finding a satisfying truth assignment for a CNF formula. In this section we briefly review some of the most well-known algorithms that are related to the local search procedures for computing stable models which are discussed in the following.

A local search algorithm for SAT begins with a random truth assignment to the variable of the propositional CNF theory, and moves in the search space by changing the value ("flipping") of one of these variables at a time. The objective function that these procedures attempt to minimize is the number of unsatisfied clauses. However, different algorithms employ different methods for selecting the variable that is flipped at each step. These algorithms can be divided into two large families, the GSAT and WSAT family. The GSAT family includes the following algorithms:

- **GSAT** [19] is the first local search algorithm for SAT. At each step it flips the variable that minimizes the total number of unsatisfied clauses.

- **GSAT-RW** [18] introduces a *random walk* step in GSAT. With probability p, it selects an unsatisfied clause c and flips one of the variables of c. With probability $1 - p$ it follows GSAT.
- **GSAT-TABU** [13] keeps a FIFO list (the tabu list) of flipped variables of fixed length tl and forbids any of the variables in the list to be flipped again. A variable that is flipped at some step can be flipped again only after tl steps.

Algorithms in the WSAT family, work in two stages. First an unsatisfied clause c is randomly selected. Then, one of the variables of c is selected and flipped according to some heuristic. These heuristics include [14]:

- **WSAT-G** with probability p (called the noise parameter) flips any variable, otherwise flips the variable that minimizes the total number of unsatisfied clauses.
- **WSAT-B** with probability p flips any variable, otherwise flips the variable that causes the smallest number of new clauses to become unsatisfied.
- **WSAT-SKC** if there is a variable p that if flipped does not cause any clauses that are currently satisfied to become unsatisfied, flip p. If no such variable exists, follow WSAT-B.

3 Randomizing Systematic Search

This section discusses a simple modification of the heuristic method that is used in the search procedure of `Smodels`. The idea was originally introduced in [8] in the context of SAT, and aims at introducing a controlled form of randomization into systematic search algorithms. `Smodels` version 2.26, implements some form of randomization through the parameters *conflicts* and *tries*. Setting parameter *conflicts* to an integer value, causes `Smodels` to terminate the search when the number of backtracks reaches the *conflicts* value. When this happens, the whole procedure starts again at the root of the search tree. The number of such restarts is determined by the value of the *tries* parameter. This technique allows `Smodels` to terminate a search that does not appear to move towards a solution.

`Smodels`, associates with every variable a heuristic value which estimates its suitability as a branching point. At each node of the search tree, the algorithm branches on the variable with the highest heuristic score, breaking ties in favor of the one that is found first. To avoid exploring the same part of the search space each time search starts from the root, `Smodels` performs a shuffling operation on the variables. Therefore, there is a probability that, at each try, the algorithm will select a different variable, provided that there is more than one with the same heuristic value.

The above technique implemented by `Smodels` may fail if the heuristic evaluation method of the algorithm tends to assign the highest heuristic score to only one or very few variables. Then, there is a high probability that `Smodels` will explore repeatedly the same regions of the search tree. To cope with these situations, we introduced the "heuristic equivalence" parameter, that has been proposed in [8], into the `Smodels` algorithm. A value H for this parameter, causes

`Smodels` to consider as equally good, all variables that have heuristic score that is not more than H-percent lower than the best score. Experimental results we report at the end of the paper suggest that, as in SAT, the heuristic equivalence parameter can lead to considerable computational savings. The implementation of the method in `Smodels` is straightforward.

4 Local Search Algorithms

In this section we present different local search algorithms for computing the stable models of a normal logic program that may contain choice rules.

4.1 The Generic Algorithm

The generic stable models local search algorithm that we introduce is identical to the local search procedure for SAT (as presented eg. in [10]). The input of the local search algorithm is a logic program P and values for the parameters $MaxTries$ and $MaxFlips$. It starts with procedure $Simplify(P, P')$ that computes the well-founded semantics [23] of the input program P, and simplifies P by fixing the values of the atoms that are assigned true or false by the well-founded semantics. This preprocessing step may reduce the size of the input logic program, and therefore speed up the computation of a stable model by the local search algorithm that takes as input the new simplified program P'. The algorithm returns a stable model or reports failure. Its description in pseudocode is as follows.

Algorithm `LSM`$(P, MaxTries, MaxFlips)$
Simplify(P, P'); $P = P'$;
for $i := 1$ **to** $MaxTries$ **do**
$M :=$ random assignment on $Atoms(P)$;
  for $j := 1$ **to** $MaxFlips$ **do**
  if M is a stable model **then return** M;
  else
    $x :=$ chooseAtom(P, M);
    $M := M$ with the truth value of x flipped;
  endif
  endfor
endfor
return "No solution found"

The critical decision we are confronted with in the new algorithm, concerns the selection of the literal that is flipped. Different instances of the *chooseAtom* procedure lead to different local search algorithms. As we noted earlier, all current propositional satisfiability local search procedures adopt the objective of minimizing the number of unsatisfied clauses. Note, that checking whether a CNF clause is satisfied is straightforward, and it only requires knowledge of the

truth value of the literals of the clause. The analogue of a clause in the case of logic programs is the rule. However, the notion of a satisfied or unsatisfied rule is less clear. We will come back to this issue in the following.

The family of local search algorithms that are presented in this paper, depart from approaches that directly associate conflicts with clauses. Stable model semantics allow us to define conflicts in terms of *atoms*. More precisely, our algorithms are based on the notion of an atom being in conflict with a value assignment M.

Definition 1. *Let P be a normal logic program and M a truth assignment to the atoms of $Atoms(P)$. An atom p* is in conflict with the truth assignment M, *if*

1. *p is assigned the value true in M and $p \not\in cl(P^M)$, or*
2. *p is assigned the value false in M and $p \in cl(P^M)$*

The *conflict set* $CFT(M, P)$ of a program P wrt an assignment M is the set of all atoms that are in conflict with M. It is easy to see if S is a stable model of P, $CFT(S, P)$ is empty.

The conflict set is the most fundamental concept in the development of local search procedures that compute stable models. All algorithms presented in this paper adopt the objective of minimizing the cardinality of conflict sets, by flipping an atom that leads to a new assignment that either decreases the total number of atoms that are in conflict with the assignment, or does not increase this number.

We define the *neighborhood* of an assignment M wrt a set of atoms $A \subseteq Atoms(P)$, denoted by $N(M, A)$, as the set of all value assignments that differ from M in the value of exactly one atom of A. If $A = Atoms(P)$, we denote $N(M, A)$ by $N(M)$.
An assignment $M' \in N(M, A)$ is called a *locally optimal assignment* wrt $N(M, A)$, if $M' \not\models M$ and $|CFT(M', P)| \leq |CFT(M'', P)|$, for all $M'' \in N(M, A)$. An assignment $M' \in N(M, A)$ is called a *minimum breaks assignment* wrt $N(M, A)$, if $M' \not\models M$ and $|CFT(M', P) - CFT(M, P)| \leq |CFT(M'', P) - CFT(M, P)|$, for all $M'' \in N(M, A)$.

4.2 The GSmodels Algorithm

The first algorithm we present is called GSmodels, and can be seen as the analogue to GSAT for the case of logic programs. Algorithm GSmodels can be realized by implementing $chooseAtom(P, M)$ of the generic algorithm, as a procedure that returns the atom that if flipped results in the largest decrease in the number of conflicting atoms. In other words, GSAT changes the truth value of an atom in an assignment M, if the resulting assignment is locally optimal wrt $N(M)$.

The main problem with the above basic version of the GSmodels algorithm is that it can be easily trapped in local minima, and the restart mechanism is the only way to escape these minima. The simplest way of mitigating this problem is by introducing noise in the procedure. This strategy is implemented through the following version of *chooseAtom*.

Procedure chooseAtom-RW(P, M)
with probability p return a random atom of $CFT(M, P)$;
with probability $1-p$ return the atom that if flipped results in a new assignment that is locally optimal wrt $N(M)$;

We call the resulting algorithm RWGSmodels. We view this algorithm as the analogue to the GSAT-RW procedure. Of course, the correspondence between the two procedures is not exact; GSAT-RW selects a random unsatisfied clause and then flips a random variable of this clause, while for RWGSmodels the notion of an unsatisfied rule has not been defined.

4.3 The WalkSmodels Algorithms

The main difficulty with the GSmodels procedure is that it needs to flip all atoms of set $Atoms(P)$, in order to find one that leads to a locally optimal assignment. This is a costly operation since there can be many atoms, and the deductive closure of the logic program has to be computed each time one of them is flipped. The WSAT procedures for propositional theories, avoid a similar problem by selecting first an unsatisfied clause and then flipping literals only from this clause. In the case of the stable model semantics, if M is the current assignment, instead of considering all atoms of $Atoms(P)$, we can restrict the selection of the atom that is flipped to the elements of the set $CFT(M, P)$, or some of its subsets. The idea of selecting a subset of the conflicting atoms as the set of candidates for flipping, seems appealing for two reasons. First, the set $CFT(M, P)$ may be large, and second, and more important, the subset selection introduces an extra level of randomization, since neighborhoods now change dynamically.

The `LSM` system implements three different approaches to the selection of the atoms that are candidates for flipping. The size of the set of candidates in all three cases is determined by the value of the *set cardinality* parameter which is supplied by the user. We denote the value of this parameter by SC.

The first approach, called *random atom set* method, is straightforward, and it has been implemented in order to verify our intuition that more elaborate techniques, as those described in the following, are indeed necessary. The random atom set method, simply selects randomly a set of atoms $S \subseteq CFT(M, P)$ of size SC. Atoms that belong to the *compute* statement of a logic program have a fixed value that can not change during the search, and therefore the set of atoms that can be flipped is $CFT(M, P) - compute(P)$. `LSM` implements this method as described below.

Procedure chooseAtomSet-R(P, M, SC, S)
if $CFT(M, P) - compute(P) = \emptyset$ then return a randomly selected subset of $Atoms(P) - compute(P)$ of size SC;
else if $|CFT(M, P) - compute(P)| < SC$ then return $CFT(M, P) - compute(P)$;
else return a set of atoms S that contains SC randomly selected elements of $CFT(M, P) - compute(P)$;

Note that it is possible that all conflicting atoms wrt the current assignment, are in the compute statement of a logic program. These cases are handled by the first line of the previous procedure, that will cause the local search algorithm to flip an atom that is not in $CFT(M, P)$.

The second atom selection procedure that is implemented in `LSM`, is called *conflict atom set* method. The main idea of this method is to associate conflicting atoms with rules that are responsible for the conflict. Changing the values of some of the atoms that occur in the bodies of these rules may resolve the conflict. In local search algorithms for propositional satisfiability, the notion of an unsatisfied clause is straightforward. In stable model semantics the notion of an unsatisfied rule is not so immediate. Consider for example a logic program that contains the rules $a \leftarrow not\ b$, and $a \leftarrow not\ c$ and the value assignment $a, b, c = T$. It is unclear whether one or both rules are unsatisfied. Moreover, a change in the value of b, which occurs only in the first rule, seems enough to satisfy both rules. Finally, while in the case of an unsatisfied CNF clause flipping one of the literals of the clause satisfies the clause, this is not necessarily the case for the rules of a logic program.

The way `LSM` associates conflicting atoms with rules is based on the idea of the *conflict rule set* that is defined below.

Definition 2. *Let M be an assignment and p an element of $Atoms(P)$ such that $p \in CFT(P, M)$. We define the* conflict rule set *$CRS(p, P, M)$ of p wrt to the assignment M in a program P as follows:*

- Case 1*: if p is assigned false in M and $p \in cl(P^M)$, define $CRS(p, P, M) = \{r \in P | head(r) = p, body(r)^+ \subseteq cl(P^M), body(r)^- \subseteq M^-\}$*
- Case 2*: if p is assigned true in M and $p \not\in cl(P^M)$, define $CRS(p, P, M) = \{r \in P | head(r) = p\}$*

The *conflict rule atom set* $CAS(p, P, M)$ of an atom p wrt to the assignment M in a program P, is defined as $CAS(p, P, M) = \cup\{body(r) | r \in CRS(p, P, M)\} \cup \{p\} - compute(P)$. The procedure that implements this selection strategy is described below.

Procedure chooseAtomSet-CAS(P, M, SC, S)
Select randomly an element p of $CFT(M, P)$ and compute $CAS(p, P, M)$;
if $CAS(p, P, M) = \emptyset$ then return a randomly selected subset S of $Atoms(P) - compute(P)$ of size SC;
else if $|CAS(p, P, M)| \leq SC$ then return $CAS(p, P, M)$;
else if $|CAS(p, P, M)| > SC$ then return a randomly selected subset S of $CAS(p, P, M)$ of size SC;

The last method for selecting a set of atoms that are candidates for flipping, is called *conflict atom bodies* method and can be seen as a procedure that combines the two methods described previously. The set of candidate atoms is formed by all atoms that are in conflict, together with the bodies of the rules in which these atoms appear in their heads. The pseudocode of this method is as follows.

Procedure chooseAtomSet-CB(P, M, SC, S)
Compute the set $S' = \cup_{p \in CFT(M,P)} CAS(p, P, M) \cup CFT(M, P) - compute(P)$;
if $S' = \emptyset$ then return a randomly selected subset S of $Atoms(P) - compute(P)$ of size SC;
else if $|S'| \leq SC$ then return S';
else if $|S'| > SC$ then return a random subset S of S' of size SC;

The set of candidate atoms that is returned by the last two procedures may contain some p such that $p \not\in CFT(M, P)$. Note that it can be the case that a locally optimal assignment is obtained by flipping an atom that does not belong to $CFT(M, P)$. Consider for example the logic program that consists of the rules $a \leftarrow not\ b$ and $b \leftarrow not\ a$, together with the set of rules $p_i \leftarrow not\ a$, for $2 < i \leq n$. Assume that the current assignment is $a = T$, $b = F$, and $p_i = T$. Note that flipping atom a causes the number of conflicts to reduce by $n - 2$.
However, `LSM` can be enforced to restrict its selection of candidate atoms to the elements of the set $CFT(M, P)$, by activating the parameter *only-conflicting*. It turns out that for some problems this restriction leads to improved performance.

Once the set of atoms that are candidates for flipping is selected, the algorithm must choose one of them that is actually flipped. The current version of `LSM` implements two different heuristics for this selection. The first, called WalkSM-G, is similar to the heuristic employed in the WSAT-G algorithm for SAT. The procedure that returns the atom that is flipped at each step is the following.

Procedure chooseAtom-WalkSM-G(P, M)
call $chooseAtomSet(P, M, CS, S)$;
with probability p do:
  flip a random atom of S;
with probability $1 - p$ do:
  flip an atom of S that leads to a locally optimal assignment wrt $N(M, S)$;

The second heuristic that selects the atom that is flipped is called WalkSM-SKC and is similar to the heuristic used in WSAT-SKC. Its pseudocode is as follows.

Procedure chooseAtom-WalkSM-SKC(P, M)
call $chooseAtomSet(P, M, SC, S)$;
if there is $b \in S$ such that if flipped no other atom becomes conflicting, flip b;
else
with probability p do:
  flip a random atom of S;
with probability $1 - p$ do:
  flip an atom of S that leads to a minimum breaks assignment wrt $N(M, S)$;

The two above *chooseAtom* procedures are parametric to the *chooseAtomSet* method they employ, and different choices for this parameter lead to local search procedures with different computational behavior.

Note that all the above algorithms form a subset of atoms that are candidates for flipping, either from atoms that belong to the current conflict set, or atoms that are *syntactically* related to the elements of this set. That is, they consider as candidates, either conflicting atoms or atoms that appear in the same rule with those that are in conflict. This feature necessitates the use of value assignments on the whole[1] set $Atoms(P)$, as explained by the following example.
Consider the program P that consists of the rules[1ex]

$$p \leftarrow q$$
$$q \leftarrow not\ b$$
$$b \leftarrow not\ c$$
$$c \leftarrow not\ b$$

and the statement $compute(\{p\})$. Let our current assignment be $M = \{b = T,\ c, q, p = F\}$, which gives $CFT(M, P) = \{p\}$. Observe that none of the atoms b and c that appear negated in P, either belongs to $CFT(M, P)$, or appears in the same rule together with some element of $CFT(M, P)$. Therefore, if the above algorithms were restricted to consider only atoms that appear negated as candidates for flipping, they would reach a deadlock.
In the GSmodels algorithm such situations can not arise, therefore GSmodels needs only consider truth assignments on the set of atoms that occur negated in the program.

Finally, we note that the current version of `LSM` also implements some tabu search techniques in the vein of GSAT-TABU. However, since the usefulness of these techniques, in their full generality, is unclear for the moment, they are not discussed further in the paper. Nevertheless, tabu lists of length 1 and 2 are used in the experiments that are presented in section 5, as they lead to improved performance in most problems.

4.4 Local Search with Choice Rules

Choice rules are one of the early extensions of the syntax of normal logic programs that were implemented in `Smodels` [21]. A choice rule r is of the form

$$\{h_1, h_2, ..., h_k\} \leftarrow a_1, a_2, ..., a_n,\ not\ b_1,\ not\ b_2, ...,\ not\ b_m$$

and can be used to express a nondeterministic choice on the atoms $\{h_1, h_2, \ldots, h_k\}$. The semantics of a choice rule is that whenever the body of the rule is satisfied by a stable model, any subset of the atoms $\{h_1, h_2, ..., h_k\}$ can be included in this model.

The local search algorithms presented in the previous section can be easily extended to accommodate choice rules. Assume that the current assignment

[1] Contrast this with the systematic search procedure of `Smodels` which assigns values only to atoms that appear negated in the program [21].

is M, and that $a_i \in cl(P^M)$ and $b_i \in M^-$ hold for a choice rule r. Then, `LSM` adds to the deductive closure only those atoms h_i that appear in the head of r, for which $h_i \in M^+$ holds. Therefore, choice rules can not give rise to conflicts.

In order to handle correctly programs with choice rules, the definition of the conflict rule set $CRS(p, P, M)$ of an atom p that appears in the head of the choice rule r, is modified as follows. Assume that $body(r)^+ \subseteq cl(P^M)$ and $body(r)^- \subseteq M^-$ hold, and furthermore $p \in cl(P^M)$ while p is assigned false in M. Clearly, r can not be responsible for this conflict, therefore r is not be included in $CRS(p, P, M)$.
Assume now that p is assigned true in M and $p \not\in cl(P^M)$. Then, r is included in $CRS(p, P, M)$ since flipping the atoms in the body of r may resolve the conflict.

We note that conflict rules are not implemented in the current version of `LSM`, except for the case of choice rules with an empty body.

5 Experimental Results

We ran some initial experiments with the new algorithms in a number of different problems. The experiments concern both the heuristic equivalence method for randomizing systematic search and the local search algorithms.

Some experiments with the randomization method of [8] are presented in Table 1. The first two rows refer to n-queens problems, while the rest to AI planning problems. Each problem was run with different values for the heuristic equivalence parameter, in the range 0 to 50, as shown in the Table 1. Depending on the hardness of each problem, different values for the *conflicts* parameter were used, and are depicted in Table 1. The *tries* parameter was set to 10 for all problems. Each experiment was repeated 3 times with different seeds. The entries of Table 1 are in the `s/c` format, where `s` denotes the number of times a solution was found, and `c` the average number of conflicts (backtracks) over all successful repetitions of the experiment. A dash denotes that no solution was found in any of the 3 runs of the experiment.

It appears that the heuristic equivalence method can substantially improve the restarts based search procedure of `Smodels`. Furthermore, the new method was able to quickly solve problems for which the standard backtracking procedure of `Smodels` requires very long run times.

Experimentation with the local search procedures is more complicated as there are many parameters involved. Table 2 presents some results for the algorithms that seem to perform best on the average. Rows `ham` correspond to hamiltonian circuit problems, `4col` to graph coloring, `queens` to n-queens problems, and finally `sat` to 3CNF SAT problems.

Each column of Table 2 corresponds to a different local search algorithm as follows. Algorithm A1 is the combination of WalkSM-G with method CB (conflict atom bodies), and A2 is the same as A1 with the difference that the atoms that are selected are conflicting atoms (parameter *only-conflicting*). Procedure A3 is WalkSM-G combined with the random atoms method (R), while A4 is

Table 1. Number of backtracks for `Smodels` for different values of the heuristic equivalence parameter

Problem	conflicts	0%	10%	20%	30%	40%	50%
`queens20`	70	-	1/140	3/197	3/25	3/2	3/1
`queens30`	70	-	-	2/214	3/1	3/10	3/4
`logistics1`	50	-	-	2/328	3/129	3/56	3/205
`logistics2`	50	-	-	3/168	2/129	3/234	3/118
`trains1`	60	-	-	1/334	3/84	2/265	3/248
`trains2`	60	-	-	-	3/149	3/170	2/257

WalkSM-G combined with conflict rule atom set (CAS). Finally, A5 is WalkSM-SKC combined with CB on conflicting atoms.

The results of Table 2 were obtained with the following parameter settings. For problems `ham`, `4col` and `queens`, the value of the set cardinality parameter SC was set to 15, the noise probability to 10% and the tabu list length to 2. For all `sat` problems the value 5 was used for SC, the noise probability was set to 15% and the tabu list length was set to 1.

Each problem was run 5 times with the *MaxFlips* parameter set to 20000. The entries of Table 2 are in the `s/c` format, where `s` denotes the number of times a solution was found, and `c` the average number of flips over all successful runs of the experiment. A dash denotes that no solution was found in any of the 5 runs of the experiment or that the corresponding algorithm is not meaningful for the particular representation of the problem that has been used.

In order to clarify the relationship between the representation of a problem and the algorithm that is used for solving that problem, we consider the encoding of SAT problems in normal logic programs. The representation of a CNF formula on the atoms $p_1, p_2, ..., p_n$ that has been used in the experiments, is similar to that described in [21], and is as follows. First, a choice rule of the form $\{p_1, p_2, ..., p_n\} \leftarrow$ is used to express the truth assignments on the atoms of the formula. Then, each clause C_i of the form $p_{p_1} \vee \ldots \vee p_{p_n} \vee \neg p_{n_1} \vee \ldots \vee \neg p_{n_m}$ translates into a rule of the form $C_i \leftarrow not\ p_{p_1}, \ldots, not\ p_{p_n}, p_{n_1}, \ldots, p_{n_m}$. Finally, the statement $compute(\{not\ C_1, not\ C_2, \ldots, not\ C_k\})$ is added, where k is the number of clauses of the CNF formula. Note that the only atoms that can be in conflict in an assignment are the atoms C_i. These atoms however can not take a truth value other than false, as determined by the *compute* statement. Therefore, a local search algorithm that restricts the atoms that are flipped only to those that are in conflict, is not applicable in this representation of SAT problems. The effect of the encoding of a problem on the performance of local search algorithms appears to be an important issue, but it is not discussed further.

When the parameter *only-conflicting* is suitably selected, it seems that the combination of WalkSM-G with the conflict atom bodies method outperforms all other combinations. Parameter settings for this algorihtm that seem to work reasonably well across a variety of problems, are set cardinality in the range 5

Table 2. Comparison of different local search procedures

Problem	A1	A2	A3	A4	A5
ham28	1/6737	5/3526	5/2203	2/9787	-
ham30	2/7695	5/4152	5/3027	3/10956	-
4col90	5/1930	2/10878	1/3032	-	3/10069
4col100	5/3458	2/9948	2/10462	1/1042	1/17817
queens20	-	5/1444	5/1607	-	5/5205
queens25	-	5/6162	5/3410	-	5/5577
queens30	-	5/7462	5/6689	-	5/6278
sat1	5/4978	-	-	-	-
sat2	5/3185	-	-	3/4251	-
sat3	4/6802	-	-	1/7516	-

to 15 and noise probability in the range 10% to 20%. Moreover the use of a tabu list of size 1 or 2 seems helpful in most problems. The combination WalkSM-G with conflict atom bodies, solved some hard instances of SAT problems that correspond to the `sat` rows of Table 2.

With the particular encoding of the propositional satisfiability problems that has been used in the experiments, the systematic search procedure of `Smodels` performed 14824 backtracks before finding a solution for problem `sat3`, and 10746 in problem `sat1`. With our representation of the n-queens problem, the systematic search algorithm of `Smodels` failed to find a solution for `queens30` within 1 hour of CPU time.

6 Conclusions and Future Work

In this paper we presented several different local search algorithms for computing a stable model of a logic program and discussed how the heuristic equivalence method can be used to randomize the systematic search algorithm of `Smodels`.

The local search algorithms we developed are mainly variants of local search procedures that have been applied successfully to the problem of propositional satisfiability. The initial experimental results reported in the paper are promising, as there are cases where the local search procedures clearly outperform the systematic algorithm. However, systematic and local search procedures should not be viewed as competing but rather as complementary. Each of them has different strengths, and therefore systems that implement both methods are capable of solving more problems than can be solved by each of them separately.

Our future work will be on improving the implementation and extending the local search procedures to more expressive rules that are included in the syntax of `Smodels` [20]. Also more intensive experimentation is needed in order to understand better the strengths and weaknesses of the local search procedures, as well as their exact relation to similar procedures for SAT. Additionally, the relationship between the algorithms of this paper and the work described in [15], [16]

and [1] needs to be studied. Finally, some ongoing experimentation indicates that the "global" nature of the stable model semantics necessitates the use of more sophisticated local search procedures when the problems that are solved are highly structured. Therefore, more advanced algorithms for local search such as those described in [5,17] will be implemented, and the combination of local and systematic search procedures [11] will be examined.

References

1. A. Bertoni, G. Grossi, A. Provetti, V. Kreinovich, and L. Tari. The prospect for answer sets computation by a genetic model. In *Proc. of the AAAI Spring 2001 Symposium on Answer Set Programming*. http://www.cs.nmsu.edu/~tson/ASP2001/, 2001. 364, 376
2. T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System description: DLV. In *Proc. of the 6th Intern. Conf. on Logic Programming and Nonmonotonic Reasoning, LPNMR-01*, LNCS 2173, pages 424–428. Springer Verlag, 2001. 363
3. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proc. of the 4th European Conference on Planning, ECP'97*, LNCS 1348, pages 169–181. Springer Verlag, 1997. 363
4. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1/2):99–111, 1999. 363
5. J. Frank. Learning short-term weights for GSAT. In *Proc. 15h Intern. Joint Conference on AI, IJCAI-97*, pages 384–391, 1997. 376
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th Intern. Conf. and Symp. on Logic Programming, ICSLP-88*, pages 1070–1080, 1988. 363
7. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991. 363
8. C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of the 15th National Conference on AI, AAAI-98*, pages 431–437, 1998. 363, 366, 373
9. K. Heljanko and I. Niemela. Bounded LTL model checking with stable models. In *Proc. of the 6th Intern. Conf. on Logic Programming and Nonmonotonic Reasoning, LPNMR'01*, LNCS 2173, pages 200–212. Springer Verlag, 2001. 363
10. H. Hoos and T. Stutzle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000. 367
11. N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Proc. of the 17th National Conference on AI, AAAI-00*, pages 169–174, 2000. 376
12. N. Leone, S. Perri, and P. Rullo. Local search techniques for disjunctive logic programs. In *6th Congress of the Italian Association for Artificial Intelligence, AI*IA 99*, LNCS 1792, pages 107–118. Springer Verlag, 2000. 364
13. B. Mazure, L. Sais, and E. Gregoire. Tabu search for SAT. In *Proc. of the 14th National Conference on AI, AAAI-97*, pages 281–285, 1997. 366
14. D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proc. of the 14th National Conference on AI, AAAI-97*, pages 321–326, 1997. 366

15. P. Nicolas, F. Saubion, and I. Stephan. GADEL: a genetic algorithm to compute default logic extensions. In *Proc. of the 14th European Conference on AI, ECAI'00*, pages 484–488. IOS Press, 2000. 364, 375
16. P. Nicolas, F. Saubion, and I. Stephan. New generation systems for non-monotonic reasoning. In *Proc. of the 6th Intern. Conf. on Logic Programming and Nonmonotonic Reasoning, LPNMR'01*, LNCS 2173, pages 309–321. Springer Verlag, 2001. 364, 375
17. D. Schuurmans and F. Southey. Local search characteristics of incomplete SAT procedures. In *Proc. of the 17th National Conference on AI, AAAI-00*, pages 297–302, 2000. 376
18. B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. of the 12th National Conference on AI, AAAI-94*, pages 337–343, 1994. 366
19. B. Selman, H. Levesque, and D. Mithcell. A new method for solving hard satisfiability porblems. In *Proc. of the 10th National Conference on AI, AAAI-92*, pages 440–446, 1992. 365
20. P. Simons. Extending the stable model semantics with more expressive rules. In *Proc. of the 5th Intern. Conf. on Logic Programming and Nonmonotonic Reasoning, LPNMR-99*, LNCS 1730, pages 305–316. Springer Verlag, 1999. 365, 375
21. P. Simons. Extending and implementing the stable model semantics. Ph.D. Thesis, Research Report 58, Helsinki University of Technology, 2000. 372, 374
22. T. Surjanen and I. Niemela. The Smodels system. In *Proc. of the 6th Intern. Conf. on Logic Programming and Nonmonotonic Reasoning, LPNMR-01*, LNCS 2173, pages 434–438. Springer Verlag, 2001. 363
23. A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991. 367

A Rewriting Method for Well-Founded Semantics with Explicit Negation

Pedro Cabalar

Dept. of Computer Science, University of Corunna, Spain
cabalar@dc.fi.udc.es

Abstract. We present a modification of Brass et al's transformation-based method for the bottom-up computation of well-founded semantics (WFS), in order to cope with explicit negation, in the sense of Alferes and Pereira's WFSX semantics. This variation consists in the simple addition of two intuitive transformations that guarantee the satisfaction of the so-called *coherence principle*: whenever an objective literal is founded, its explicit negation must be unfounded. The main contribution is the proof of soundness and completeness of the resulting method with respect to WFSX. Additionally, by a direct inspection on the method, we immediately obtain results that help to clarify the comparison between WFSX and regular WFS when dealing with explicit negation.

1 Introduction

Logic Programming (LP) has become one of the most popular tools for practical nonmonotonic reasoning, thanks to the use of declarative semantics, particularly, *stable models* [7] and *well-founded semantics* (WFS) [13]. This success is probably due to two important facts: (1) the availability of efficient algorithms and implementations for computing these semantics; and (2), the evolution of the basic LP paradigm to allow a more flexible knowledge representation. Of course, the progress in these two directions has not been simultaneous: we typically face new LP extensions for which the current implementations are not applicable.

For efficiency purposes, the improvement of inference methods for WFS has become interesting, not only when using WFS as a basic semantics, but also for stable models checkers, since they usually involve intermediate computations of the well-founded model. This interest motivated the research line followed by Brass et al. [3], who developed a method that improves the efficiency of the original *alternated fixpoint procedure* introduced by van Gelder [12]. Their method relies on the successive application of simple program transformations, until an equivalent non-reducible program (the so-called *program remainder*) is obtained.

Brass et al's method, however, was not thought for Extended Logic Programming (ELP), i.e., logic programs with explicit negation. The introduction of ELP comes from the need for distinguishing between *default negation*, '*not p*' (that is, we fail to prove p) and *explicit negation* ' $\overline{p}$ ' (that is, we assert p

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 378–392, 2002.
© Springer-Verlag Berlin Heidelberg 2002

to be explicitly false). In the case of stable models, the required modification is straightforward: we just rule out stable models containing any pair $\{p, \overline{p}\}$ (where $\overline{p}$ is simply treated as a regular atom). These stable models receive the name of *answer sets* [8]. Unfortunately, in the case of WFS, this contradiction rejection is not enough, as observed by Alferes and Pereira [9,1]. More concretely, one would expect that whenever an atom p is explicitly false, $\overline{p}$, its default negation, *not* p, should hold (*coherence principle*), something not guaranteed in the resulting well-founded model, which may leave *not* p undefined. To incorporate the coherence principle into WFS, Alferes and Pereira proposed a variation called WFSX (*well-founded semantics with explicit negation*).

In this paper we study how to update the program remainder method to compute WFSX. We will show that the update is, in fact, quite simple and natural, consisting in the addition of two intuitive transformations to the already existing ones for computing WFS. As a result, we prove that WFS obtains less or equal information (i.e., defined atoms) than WFSX. The modified method becomes especially interesting for the use of WFSX as an underlying semantics, but it could also be applied to improve the computation of answer sets, since it obtains more information in each intermediate computation of the well-founded model.

The paper is organized as follows. The next section contains a brief review of basic LP definitions and WFS. Sections 3 and 4 respectively describe Brass et al's transformation method and Alferes and Pereira's WFSX. Section 5 presents the proposed variation of Brass et al's method, explaining the main results. Finally, Section 6 concludes the paper. Proofs for theorems in Section 5 have been collected in Appendix A.

2 Basic Definitions

The syntax of logic programs is defined starting from a (possibly infinite) set of ground atoms $\mathcal{H}$ called the *Herbrand base*. We assume that all the variables have been previously replaced by their possible ground instances. We will use lower-case letters $a, b, c, \ldots, p, q, \ldots$ to denote atoms from $\mathcal{H}$. A *program literal* is either an atom $a \in \mathcal{H}$ or its default negation *not* a (which is called a *default literal*). A *normal logic program* is a (possibly infinite) set of *rules* of shape:

$$H \leftarrow B_1, \ldots, B_n \tag{1}$$

where $n \geq 0$, H is an atom and the $B_i's$ are program literals. Given a rule r like (1), we respectively define its *head* and *body* as $head(r) = H$ and $body(r) = \{B_1, \ldots, B_n\}$. When $n = 0$, we usually write 'H' to stand for '$H \leftarrow$', and say that H is a program *fact*. For any normal logic program P, the sets $facts(P)$ and $heads(P)$ respectively contain the program facts and the heads of all the rules in P. Consequently, $facts(P) \subseteq heads(P)$. A normal logic program is said to be *positive* (or *definite*) when it does not contain any default literal. Another interesting type of logic programs are those not containing any kind of cyclic dependence. Program P is said to be *hierarchical* (or *acyclic*) when all its

atoms can be arranged in levels, i.e., there exists an integer mapping $\lambda : \mathcal{H} \to \mathcal{Z}$ satisfying $\lambda(head(r)) < \lambda(b_i)$, for any rule r and each atom b_i occurring (possibly negated) in $body(r)$.

A (2-valued) *interpretation* M is defined as any subset of $\mathcal{H}$, $M \subseteq \mathcal{H}$. It can also be seen as a function $M : \mathcal{H} \to \{\mathtt{t}, \mathtt{f}\}$ mapping a truth value for each atom in $\mathcal{H}$, so that $M(a) = \mathtt{t}$ iff $a \in M$. The interpretation can be extended to provide a valuation for any formula ϕ, so that $M(\phi)$ follows the standard propositional definitions, where '*not* ', ',' and '$\leftarrow$' represent classical negation, conjunction and material implication, respectively. When $M(\phi) = \mathtt{t}$ we say that M *satisfies* ϕ. An interpretation is said to be a *model* of a program P iff it satisfies all its rules.

As shown in [11], any positive logic program P has a *least model* (with respect to set inclusion), which corresponds to the least fixpoint of the monotonic operator T_P:

$$T_P(M) = \{c \mid (c \leftarrow a_1, \ldots, a_n) \in P, \text{ and } a_i \in M \text{ for all } i \in [1, n]\}$$

As T_P is monotonic, the Knaster and Tarski's theorem [10] applies, and the least fixpoint is computable by iteration of T_P, starting from the smallest interpretation $\emptyset$.

We define the *reduct* of a normal logic program P with respect to an interpretation M, written P^M, as:

$$P^M = \{(c \leftarrow a_1, \ldots, a_n) \mid (c \leftarrow a_1, \ldots, a_n, not\, b_1, \ldots, not\, b_m) \in P, \text{ and } b_i \not\in M \text{ for all } i \in [1, m]\}$$

Since P^M is a positive program, it has a least model we will denote as $\Gamma_P(M)$, or simply $\Gamma(M)$ when there is no ambiguity. The fixpoints of Γ receive the name of *stable models*.

The well-founded model will have the shape of a 3-valued or *partial interpretation* M, formally defined as a pair (M^+, M^-) of disjoint sets of atoms. The sets M^+, M^- and $\mathcal{H} - (M^+ \cup M^-)$ represent the *true*, *false* and *undefined* atoms, respectively. Given M^-, we will usually refer to its complementary set $\mathcal{H} - M^-$, that is, the nonnegative atoms. It is clear that, since M^+ and M^- are disjoint, $M^+ \subseteq \mathcal{H} - M^-$. We say that M is *complete* iff $M^+ = \mathcal{H} - M^-$, i.e., $\mathcal{H} = M^+ \cup M^-$. Besides, two partial interpretations can be compared with respect to their amount of information (i.e., defined atoms):

Definition 1. (Information or Fitting's ordering, $\leq_F$)
We say that interpretation $M_1 = (M_1^+, M_1^-)$ *contains* less information *than interpretation* $M_2 = (M_2^+, M_2^-)$*, denoted as* $M_1 \leq_F M_2$*, iff* $M_1^+ \subseteq M_2^+$ *and* $M_1^- \subseteq M_2^-$.

The characterization of WFS relies on the fact that the Γ operator is antimonotonic, and so, Γ^2, that is, Γ applied twice, results to be monotonic. In this way, we may apply again Knaster-Tarski's theorem for Γ^2: there exists a least (resp. greatest) fixpoint, $lfp(\Gamma^2)$ (resp. $gfp(\Gamma^2)$), which is computable by iteration on the least set of atoms $\emptyset$ (resp. the greatest set of atoms $\mathcal{H}$).

Moreover, each fixpoint of this pair can be computed in terms of the other: $gfp(\Gamma^2) = \Gamma(lfp(\Gamma^2))$ and $lfp(\Gamma^2) = \Gamma(gfp(\Gamma^2))$.

Definition 2. (Well-founded model (WFM))
The well-founded model *(WFM) of a normal logic program P corresponds to the 3-valued interpretation:* $\big(lfp(\Gamma^2), \mathcal{H} - gfp(\Gamma^2)\big)$

The relation to stable models is clarified by the following well-known properties:

Proposition 1. *Let W be the WFM of a normal logic program P. For any stable model M of P, we have that $W \leq_F (M, \mathcal{H} - M)$.*

Proposition 2. *Let $W = (W^+, W^-)$ be the WFM of a normal logic program P. If W is complete, i.e. $W^+ = \mathcal{H} - W^-$, then W^+ is the unique stable model of P.*

3 Brass et al's Method

Brass et al's method can be introduced by initially identifying what we will call the *trivial interpretation* of a normal logic program.

Definition 3. (Trivial Interpretation)
The trivial interpretation *of a normal logic program P is the 3-valued interpretation:* $\big(facts(P), \mathcal{H} - heads(P)\big)$.

That is, the trivial interpretation makes true all the atoms that occur as facts in P and makes false all the atoms that are not head of any rule in P. The method is mainly based on successively simplifying rules that deal with atoms with a defined truth value in the trivial interpretation. If P is the current program, then we define the transformations:

1. *Positive reduction* $\xrightarrow{\texttt{P}}$ for any rule r, delete any literal $(not\ p) \in body(r)$ such that $p \notin heads(P)$.
2. *Negative reduction* $\xrightarrow{\texttt{N}}$ delete any rule r containing some $(not\ p) \in body(r)$ with $p \in facts(P)$.
3. *Success* $\xrightarrow{\texttt{S}}$ for any rule r, delete any literal $p \in body(r)$ such that $p \in facts(P)$.
4. *Failure* $\xrightarrow{\texttt{F}}$ delete any rule r containing some $p \in body(r)$ with $p \notin heads(P)$.
5. *Positive loop detection* $\xrightarrow{\texttt{L}}$ delete any rule r containing some $p \in body(r)$ such that $p \notin \Gamma(\emptyset)$.

Proposition 3. (See theorem 4.17 in [3])
The transformations {P,N,S,F,L} *are sound w.r.t. WFS and provide a confluent calculus which is strongly terminating. Furthermore, let P be the final program where no new transformation is applicable. This program is called the* program remainder. *Then, the WFM of P corresponds to its trivial interpretation $(facts(P), \mathcal{H} - heads(P))$.*

It is easy to see that the first four transformations just simplify rules dealing with atoms that are known in the trivial interpretation (their truth value is not undefined). As an interesting additional result, (theorem 4.9 in [3]) the exhaustive application of these four rules allows obtaining the *Fitting's model* [6] of a normal logic program. In this way, the fifth transformation, `L`, can be seen as the real "contribution" of WFS with respect to Fitting's semantics. As it is well known, Fitting's model may yield undefined atoms because of self-supported positive cycles. For instance, the Fitting's model for the simple program $\{p \leftarrow p\}$ would leave p undefined instead of false. In order to avoid this behavior, transformation `L` adopts an optimistic point of view: we compute the consequences of the program assuming all the default literals to be true (this is the real meaning of $\Gamma(\emptyset)$). Atoms that are not obtained by this procedure will always be false (since we had assumed the most optimistic case for default negation).

Let us see a simple example. Consider the program P_1:

$$\begin{array}{lll} a \leftarrow not\ b, c & d \leftarrow not\ g, e & g \leftarrow not\ c \\ b \leftarrow not\ a & e \leftarrow not\ g, d & h \leftarrow\ g \\ c & f \leftarrow not\ d & \end{array}$$

Since $c \in facts(P_1)$ we apply success to the rule for a and negative reduction to the rule for g:

$$\begin{array}{lll} a \leftarrow not\ b & d \leftarrow not\ g, e & \\ b \leftarrow not\ a & e \leftarrow not\ g, d & h \leftarrow\ g \\ c & f \leftarrow not\ d & \end{array}$$

Now, as g is not head of any rule, we apply failure to the rule for h, and positive reduction to the rules for d and e:

$$\begin{array}{ll} a \leftarrow not\ b & d \leftarrow\ e \\ b \leftarrow not\ a & e \leftarrow\ d \\ c & f \leftarrow not\ d \end{array}$$

At this point, none of the four transformations {`P`,`N`,`S`,`F`} is applicable. Fitting's model simply corresponds to the trivial interpretation of the program above: $(\{c\}, \mathcal{H} - \{a, b, c, d, e, f\})$, that is $(\{c\}, \{g, h\})$. To obtain the WFM, however, we must also consider rule `L`. Thus, we compute $\Gamma(\emptyset)$, i.e., the least model of the program:

$$\begin{array}{ll} a & d \leftarrow\ e \\ b & e \leftarrow\ d \\ c & f \end{array}$$

which clearly corresponds to $\{a, b, c, f\}$. Then, as neither d nor e belong to this model, we delete the rules containing those atoms in their bodies:

$$\begin{array}{ll} a \leftarrow not\ b & c \\ b \leftarrow not\ a & f \leftarrow not\ d \end{array}$$

Finally, as d is not head of any rule, we can apply positive reduction to the rule for f:

$$\begin{array}{ll} a \leftarrow not\ b & c \\ b \leftarrow not\ a & f \end{array}$$

Now, no new transformation is applicable, and so, the WFM of P_1 is $(\{c, f\}, \mathcal{H} - \{a, b, c, f\}) = (\{c, f\}, \{d, e, g, h\})$.

4 WFSX and the Coherence Problem

As explained before, when using LP as a nonmonotonic reasoning tool, we usually need to incorporate a second type of negation to allow representing explicitly false facts. The addition of a second negation (named in different works as *classical*, *explicit* or *strong* negation) leads to the so called *extended logic programming* (ELP) [8,9,1,2].

We will handle now two types of atoms: 'p', to represent that p has value *true*; and '$\overline{p}$', to represent[1] that p has value *false*. Normal logic programs dealing with this extended signature receive the name of *extended logic programs*. It is usual to call *objective literal*, denoted as L, to either p or $\overline{p}$ (that is, in our terminology, to any atom) and *default literal* to any *not* L. Besides, we will use the notation $\overline{L}$ to stand for the complementary objective literal of L, assuming $\overline{\overline{p}} \stackrel{\text{def}}{=} p$. Furthermore, given a set of objective literals M, we denote $\overline{M}$ to stand for their complementary literals: $\overline{M} \stackrel{\text{def}}{=} \{\overline{p} \mid p \in M\}$.

As explained in the introduction, the extension of the stable models semantics for ELP is extremely simple. We rule out the stable models containing any pair $\{p, \overline{p}\}$, that is, we take charge of explicit contradictions. We usually talk about *answer sets* [8] when referring to these (non-contradictory) stable models of extended logic programs.

At a first glimpse, it seems that something similar can be done for the WFS, considering the program to be *contradictory* when the WFM makes true both p and $\overline{p}$. However, as pointed out in [9,1], a direct application of this method may lead to counterintuitive results. To understand the problem, it must be first noted that we handle now more possible epistemic states for a given atom. Rather than saying that L is *true* (when $L \in M^+$) or that it is *false* (when $L \in M^-$), we will say instead that it is *founded* or *unfounded*, respectively. In this way, we may distinguish between being *unknown* (that is, both $p \in M^-$ and $\overline{p} \in M^-$ are unfounded) and being *undefined*, which means that for some truth value of p we cannot establish whether it is founded or not ($p \not\in M^+ \cup M^-$ or $\overline{p} \not\in M^+ \cup M^-$).

In principle, we may have that p is undefined and $\overline{p}$ defined, or vice versa. However, it seems that there should exist a connection between complementary objective literals: when $L \in M^+$ is founded, we should have $\overline{L} \in M^-$ unfounded. Unfortunately, this property, called in [9] the *coherence principle*, is not satisfied

[1] The usual notation of ELP for $\overline{p}$ is $\neg p$. However, we use here the former to emphasize the view of $\overline{p}$ as an atom, so that we can compare to regular WFS.

by the usual definition of WFS. The typical counterexample is the program P_2:

$$p \leftarrow not\ q$$
$$q \leftarrow not\ p$$
$$\overline{p}$$

Intuitively, as we know that $\overline{p}$ is founded, its default negation $not\ p$ should be immediately true, making q founded. That is, we should obtain the complete model $(\{\overline{p}, q\}, \{p, \overline{q}\})$, with $\overline{p}$ and q founded and their complements unfounded. However, it is easy to see that the WFM of P_2 is $(\{\overline{p}\}, \{\overline{q}\})$, which leaves both p and q undefined. The reason for this is that WFS does not provide any connection between p and $\overline{p}$ and so, we are not able to establish that the default literal $not\ p$ should be true when p is foundedly false.

To overcome this difficulty, Alferes and Pereira introduced a variation of WFS called WFSX (Well-Founded Semantics with Explicit Negation). For simplicity sake, we will just provide the iterative method to compute the WFM under WFSX semantics, which relies on a variation of the application of Γ^2. Most of the properties presented here have been directly extracted from [1].

Let r be a rule $H \leftarrow B$ of an extended normal logic program. By r_s we denote the seminormal version of r:

$$r_s \stackrel{\text{def}}{=} \quad H \leftarrow B, not\ \overline{H}$$

Given a extended normal program P, we write P_s to stand for the seminormal version of P:

$$P_s \stackrel{\text{def}}{=} \{r_s \mid r \in P\}$$

For any set of atoms M and any fixed program P, we write $\Gamma_s(M)$ to denote the least model of P_s^M. By convention, we consider the function Γ_s *not defined* for a contradictory M, i.e., M containing both p and $\overline{p}$. A program is *contradictory* in WFSX iff it has no fixpoints for $\Gamma\Gamma_s$.

Proposition 4. *For non-contradictory programs, there exists a least fixpoint of $\Gamma\Gamma_s$, denoted as $lfp(\Gamma\Gamma_s)$.*

The combined function $\Gamma\Gamma_s$ is monotonic (on inclusion of sets of atoms), and so, its least fixpoint (when defined) can be computed by iteration on the least possible set of atoms, $\emptyset$. This is usually denoted as $\Gamma\Gamma_s \uparrow (\emptyset)$. The well-founded model is then defined in terms of $lfp(\Gamma\Gamma_s)$ as follows:

Definition 4. (WFSX's Well-Founded Model)
The well-founded model *(WFM) of a (non-contradictory) extended logic program under the WFSX semantics corresponds to the 3-valued interpretation:*

$$\big(lfp(\Gamma\Gamma_s),\ \mathcal{H} - \Gamma_s(lfp(\Gamma\Gamma_s))\big)$$

It is interesting to note that the iteration of $\Gamma\Gamma_s$ may also be used to detect contradictory programs, as stated by the following property:

Proposition 5. *If, for some program P, the iteration of $\Gamma\Gamma_s \uparrow (\emptyset)$ reaches an interpretation that contains both p and $\overline{p}$, then the program P is contradictory in WFSX.*

Finally, another important property (proved in theorem 4.3.6 in [1]) is that WFSX actually generalizes WFS:

Proposition 6. *For programs without explicit negation, WFSX coincides with WFS.*

Unfortunately, this property does not help to establish the differences between WFS and WFSX when the program actually contains explicit negation. We will see how the WFSX extension of Brass et al's program remainder method will also be helpful to this purpose, showing that, in fact, WFSX obtains more or equal information than WFS.

5 A Rewriting Method for Computing WFSX

The update of Brass et al's method for WFSX is very simple and natural. We will just incorporate the following two new transformations:

6. *Coherence failure* $\overset{\mathtt{C}}{\longrightarrow}$ delete any rule r containing $\overline{L} \in body(r)$ such that $L \in facts(P)$.
7. *Coherence reduction* $\overset{\mathtt{R}}{\longrightarrow}$ for any rule r, delete any default literal $(not\ \overline{L}) \in body(r)$ such that $L \in facts(P)$.

Notice how, in both cases, we simplify rules dealing with some $\overline{L}$ provided that L is trivially founded (it is a fact). In such a case, the coherence reduction, R, transforms $not\ \overline{p}$ into true. In fact, this transformation is the direct implementation of the coherence principle: default negation follows from explicit negation. As for transformation C, it allows replacing $\overline{p}$ by false, momentarily assuming that the program will be non-contradictory. As we will show later, even when this assumption is not eventually satisfied, the rewriting method (including these two new rules) is still capable of detecting contradiction.

We will also need to modify the definition of trivial interpretation:

Definition 5. (Trivial Interpretation of an Extended Logic Program) *Let P be an extended logic program not containing contradictory facts. The* trivial interpretation *of P is the 3-valued interpretation:*
$\left(facts(P), (\mathcal{H} - heads(P)) \cup \overline{facts(P)}\right)$

In other words, we consider now as false not only objective literals L that are not heads, but also those ones for which $\overline{L}$ is a fact. As an example, consider the program P_3:

$$\begin{array}{ll} a \leftarrow not\ b & \overline{p} \leftarrow\ b \\ b \leftarrow not\ a & p \end{array}$$

This program cannot be further transformed using {P,N,S,F,L}. Therefore, its WFM would be $(\{p\}, \{\overline{a}, \overline{b}\})$, which leaves a, b and $\overline{p}$ undefined. The trivial interpretation would contain, in this case, more information: $(\{p\}, \{\overline{a}, \overline{b}, \overline{p}\})$. It further considers $\overline{p}$ unfounded because p is a fact. The interest of the trivial interpretation is clarified by the following theorem:

Theorem 1. *Let P be a non-contradictory extended logic program, W its WFM (under WFSX) and U its trivial interpretation. Then $U \leq_F W$.*

That is, the trivial interpretation contain less or equal information than the WFM (under WFSX).

We prove now that the whole set of transformations, {P,N,S,F,L,C,R} are sound with respect to WFSX, that is, all the programs in a transformations chain either have the same WFM or are contradictory. To this aim, we provide first a lemma that establishes that the fixpoints for $\Gamma\Gamma_s$ remain unchanged after each transformation. We introduce here a remark on notation. When $P \stackrel{\mathtt{x}}{\longmapsto} P'$ with some transformation rule x, we write Γ' and Γ'_s to express that these functions implicitly correspond to the transformed program P', instead of P.

Lemma 1. *For any transformation $\stackrel{\mathtt{x}}{\longmapsto}$ with $\mathtt{x} \in \{\mathtt{P,N,S,F,L,C,R}\}$, if $P \stackrel{\mathtt{x}}{\longmapsto} P'$ then:*

(a) if $M = \Gamma\Gamma_s(M)$ then $\Gamma_s(M) = \Gamma'_s(M)$ and $M = \Gamma'\Gamma'_s(M)$.
(b) and vice versa, if $M = \Gamma'\Gamma'_s(M)$ then $\Gamma'_s(M) = \Gamma_s(M)$ and $M = \Gamma\Gamma_s(M)$.

Using this lemma, the result of soundness for all the transformations is almost straightforward.

Theorem 2. *The transformations $\stackrel{\mathtt{x}}{\longmapsto}$, with $\mathtt{x} \in \{\mathtt{P,N,S,F,L,C,R}\}$ are sound w.r.t. WFSX. In other words, if a program P has a WFM then any resulting P', $P \stackrel{\mathtt{x}}{\longmapsto} P'$ has the same WFM. Otherwise, if P has no WFM then P' has no WFM.*

Note that this theorem just points out that the transformations do not alter the final WFM (if there exists so), but it does not specify how to obtain this WFM. Before going further, we can already use this result to compare WFSX to WFS. As the transformations {P,N,S,F,L} used for WFS are a subset of the ones we have just proved to be sound for WFSX, we immediately get that:

Theorem 3. *Let P be any extended normal logic program and let W be its WFM under WFS. Then:*

(i) If W is contradictory (it makes true both some p and $\overline{p}$) then P is contradictory in WFSX.
(ii) If the program P has a WFM, X, under WFSX then $W \leq_F X$.

Note that the opposite for (i) does not hold, that is, we may have a program which has a non-contradictory WFM in WFS but has no solution in WFSX. As a simple counterexample, consider the program P_4:

$$a \leftarrow not\ a$$
$$\overline{a}$$

It is easy to see that in WFS, the WFM is not contradictory: we get $\overline{a}$ founded while a becomes *undefined* (remember that there is no connection between both atoms). When we move to WFSX, however, the above program would still be transformable (by coherence reduction) into the pair of facts a and $\overline{a}$, which are trivially contradictory. So, the program has no WFM under WFSX.

Theorem 3 also leads to the less general, but also useful result:

Corollary 1. *Let W be the WFM under WFS for some program P, and let W be non-contradictory and complete. Then, W is also the WFM of P under WFSX.*

To end up with the comparison, as for any hierarchical program, WFS leads to a complete WFM, we also obtain:

Corollary 2. *WFS and WFSX coincide for any hierarchical program.*

As we did for WFS, we say that a program P is the *program remainder* when no further transformation in $\{\mathtt{P},\mathtt{N},\mathtt{S},\mathtt{F},\mathtt{L},\mathtt{C},\mathtt{R}\}$ is applicable.

Theorem 4. (Main Result)
Let P be the program remainder (of a possibly empty chain of transformations) and let P be free of contradictory facts. Then, the trivial interpretation of P is its WFM under WFSX.

The proof of the previous theorem (see appendix) makes use of the premise of non-applicability of all the transformations *excepting* the failure F. This means that this transformation is actually redundant. In fact, it is easy to see that failure is a particular case of positive loop detection L. Both transformations delete rules containing positive literals that satisfy a given condition which in F is stronger than in L: $L\ /\!\!\in heads(P) \Rightarrow L\ /\!\!\in \Gamma(\emptyset)$. Nevertheless, maintaining transformation F is interesting because of a pair of reasons. On the one hand, its computation means a considerably lower cost than loop detection and so, it may imply an efficiency improvement in many cases. On the other hand, it is interesting from the theoretical point of view, since, as we had seen, the subset of transformations $\{\mathtt{P},\mathtt{N},\mathtt{S},\mathtt{F}\}$ completely establish the Fitting's model of the program. Furthermore, we could even think about the extension of Fitting's semantics to cope with the coherence principle. This can be simply done by considering the set of transformations $\{\mathtt{P},\mathtt{N},\mathtt{S},\mathtt{F},\mathtt{C},\mathtt{R}\}$, that is, the ones for WFSX, excepting positive loop detection. A topic for future work could be to characterize the resulting semantics with a fixpoint definition or a models selection criterion.

6 Conclusion

We have shown how to extend Brass et al's transformation-based bottom-up computation of well-founded semantics (WFS) to cope with explicit negation, in the sense of Alferes and Pereira's WFSX semantics. The extension consists in adding two simple transformations to the ones already defined by Brass et al. for regular WFS. As expected, the additional transformations are directly related to the so-called *coherence principle* so that, whenever an objective literal L is founded, we must consider that its complementary literal $\overline{L}$ (i.e., its explicit negation) is unfounded.

The final method can be used for an efficient bottom-up computation of WFSX and it could be even applied to improve the efficiency of answer sets provers, due to their intermediate use of WFS. Future work could include a practical assessment in this sense. An specialized version of this method has been applied and implemented for the use of WFSX as an underlying semantics for causal representation of action domains [5,4].

References

1. J. J. Alferes. *Semantics of Logic Programs with Explicit Negation*. PhD thesis, Facultade de Ciências e Tecnologia, Universidade Nova de Lisboa, 1993. 379, 383, 384, 385
2. J. J. Alferes, L. M. Pereira, and T. C. Przymusinski. 'classical' negation in non-monotonic reasoning and logic programming. *Journal of Automated Reasoning*, 20(1):107–142, 1998. 383
3. S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming*, to appear, 2001. (Draft version available at `http://www.cs.man.ac.uk/~jdix/Papers/01_TPLP.ps.gz`). 378, 381, 382
4. P. Cabalar. *Pertinence for causal representation of action domains*. PhD thesis, Facultade de Informática, Universidade da Coruña, 2001. 388
5. P. Cabalar, M. Cabarcos, and R. P. Otero. PAL: Pertinence action language. In *Proceedings of the 8th Intl. Workshop on Non-Monotonic Reasoning NMR'2000 (Collocated with KR'2000)*, Breckenridge, Colorado, USA, april 2000. (`http://xxx.lanl.gov/abs/cs.AI/0003048`). 388
6. M. Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985. 382
7. M. Gelfond and V. Lifschitz. The stable models semantics for logic programming. In *Proc. of the 5th Intl. Conf. on Logic Programming*, pages 1070–1080, 1988. 378
8. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991. 379, 383
9. L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In *Proceedings of the European Conference on Artificial Intelligence (ECAI'92)*, pages 102–106, Montreal, Canada, 1992. John Wiley & Sons. 379, 383
10. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955. 380

11. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976. 380
12. A. van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computing and System Sciences*, 47(1):185–221, 1993. 378
13. A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991. 378

Appendix A. Proofs of Theorems

Proof. **(theorem 1)** Let us call $W = (W^+, W^-)$ and $U = (U^+, U^-)$. By definition of $\leq_F$, we must prove $U^+ \subseteq W^+$ and $U^- \subseteq W^-$. Consider first any $L \in U^+ = facts(P)$. The result of applying Γ will always contain the fact L. By definition, $W^+ = \Gamma\Gamma_s(W^+)$, and so $L \in W^+$. Consider now $L \in U^-$. By definition, either $L \not/ heads(P)$ or $L \in \overline{facts(P)}$, i.e., $\overline{L} \in facts(P)$. On the one hand, if $L \not/ heads(P)$, the result of applying both Γ and Γ_s cannot contain L. Then, $L \not/ \Gamma_s(W^+)$, i.e. $L \in \mathcal{H} - \Gamma_s(W^+)$ which, by definition, is W^-. On the other hand, if $\overline{L} \in facts(P)$, as $facts(P) \subseteq W^+$, then $\overline{L} \in W^+$. But then, the reduct $P_s^{W^+}$ will not contain any rule with L as head (in the seminormal program P_s, these rules contains *not* $\overline{L}$ in their bodies). As a result, $L \not/ \Gamma_s(W^+)$ which means that $L \in \mathcal{H} - \Gamma_s(W^+)$ i.e. $L \in W^-$.

Proof. **(lemma 1)**

$\overset{\texttt{F}}{\longrightarrow}$

As p is not head of any rule in P, the same applies for P_s, P' and P'_s. Thus, for any of these programs Q and for any interpretation M, when iterating $T_{Q^M} \uparrow (\emptyset)$, p is never obtained and so, any rule with p in the body is never used. Then, it can be deleted without varying the result. This directly implies that $\Gamma(M) = \Gamma'(M)$ and $\Gamma_s(M) = \Gamma'_s(M)$, for any M and so, the proofs for (a) and (b) become trivial.

$\overset{\texttt{L}}{\longrightarrow}$

We will similarly show that, for any M and any $Q \in \{P, P_s, P', P'_s\}$: $p \not/ T_{Q^M} \uparrow (\emptyset)$. As $p \not/ \Gamma(\emptyset)$ we immediately get that p cannot belong to any application of $\Gamma(M)$, because the resulting reduct is a subset: $P^M \subseteq P^\emptyset$. Besides, as $P^\emptyset = P_s^\emptyset$, we have $p \not/ \Gamma_s(\emptyset)$, and so p cannot belong to any $\Gamma_s(M)$ since again $P_s^M \subseteq P_s^\emptyset$. Finally, for Γ' and Γ'_s it suffices to see that $P' \subseteq P$ and $P'_s \subseteq P_s$. Then, the rules with p in the body are never used during the iteration $T_{Q^M} \uparrow (\emptyset)$ and so $\Gamma(M) = \Gamma'(M)$ and $\Gamma_s(M) = \Gamma'_s(M)$ for any M, being the proofs for (a) and (b) directly trivial.

$\overset{\texttt{P}}{\longrightarrow}$

As proved in $\overset{\texttt{F}}{\longrightarrow}$ since p is not head, it cannot belong to any application of $\Gamma, \Gamma_s, \Gamma'$ or Γ'_s. Besides, for any interpretation M such that $p \not\in M$, it is easy to see that $P^M = P'^M$ and $P_s^M = P_s'^M$. Let us prove first (a). For any fixpoint $M = \Gamma\Gamma_s(M)$ we have that, as M is the result of applying Γ, $p \not\in M$. But then, $\Gamma_s(M) = \Gamma'_s(M)$ (which is the first consequent of (a)), and in its turn,

$p \not\in \Gamma_s(M)$. Finally, this means that $M = \Gamma\Gamma_s(M) = \Gamma'\Gamma_s(M) = \Gamma'\Gamma'_s(M)$, that is M is fixpoint of $\Gamma'\Gamma'_s$. The proof for (b) is completely analogous.

$\mapsto$ S

Notice first that, for any M, $\Gamma(M) = \Gamma'(M)$ and $p \in \Gamma(M)$, because p is a fact in P, and so, it will always valuated as true when applying T_P (resp. $T_{P'}$). Second, we show now that for any M such that $p \in M$ and $\overline{p} \not\in M$, $\Gamma_s(M) = \Gamma'_s(M)$. The fact p occurs as the seminormal rule $p \leftarrow not\ p$ both in P_s and in P'_s, but in P_s^M and $P_s'^M$, this rule will become again the original fact p. As a result, deleting p from the rules will not vary the final outcome, i.e., $\Gamma_s(M) = \Gamma'_s(M)$. Besides, in P_s^M and $P_s'^M$ all the rules for $\overline{p}$ will be deleted (they contain in their bodies $not\ p$). This means that, additionally, $\overline{p} \not\in \Gamma_s(M)$.

Now, we prove (a): let $M = \Gamma\Gamma_s(M)$. Then, as M is the result of applying Γ then $p \in M$. But, at the same time, as Γ_s is defined for M, $\overline{p} \not\in M$. Therefore, we can apply the previous results, $\Gamma_s(M) = \Gamma'_s(M)$ (which is the first consequent of (a)). Let us call J to $\Gamma_s(M)$. Then, as we had seen, $\Gamma(J) = \Gamma'(J)$, i.e., $\Gamma\Gamma_s(M) = \Gamma'\Gamma_s(M) = \Gamma'\Gamma'_s(M)$. The proof for (b) is again analogous.

$\mapsto$ N

First, note that for any M with $p \in M$, $P^M = P'^M$ and $P_s^M = P_s'^M$. Then, for proving (a), let $M = \Gamma\Gamma_s(M)$. As before, since p is a fact in P and M is the result of applying Γ, we get $p \in M$, and $\overline{p} \not\in M$ (otherwise $\Gamma_s(M)$ would not be defined). Therefore $P_s^M = P_s'^M$ and $\Gamma_s(M) = \Gamma'_s(M)$ (the first part of (a)). Now note that the fact p in P becomes the seminormal rule $p \leftarrow not\ \overline{p}$ in P_s. However, in the reduct P_s^M (which is equal to $P_s'^M$) this rule becomes again the fact p, because $\overline{p} \not\in M$. This means that $p \in \Gamma_s(M)$, and so, $P^{\Gamma_s(M)} = P'^{\Gamma_s(M)}$. It follows that $\Gamma\Gamma_s(M) = \Gamma'\Gamma_s(M) = \Gamma'\Gamma'_s(M)$. As always, the proof for (b) is analogous.

$\mapsto$ C

Again, for any M with $p \in M$, in the reducts P_s^M and $P_s'^M$ all the rules with $\overline{p}$ as head are deleted (as they are seminormal, they contain $not\ p$ in the body). As a result, $\overline{p}$ is never added when iterating the direct consequences operator, and so $\Gamma_s(M) = \Gamma'_s(M)$. Now, consider the proof for (a). If $M = \Gamma\Gamma_s(M)$, we have $p \in M$ (because of p being a fact and M the result of Γ) and so, the previous result is applicable: $\Gamma_s(M) = \Gamma'_s(M)$, which is the first part of (a). Now, as $\Gamma_s(M)$ is defined, we get that $\overline{p} \not\in M$. But this means that when iterating direct consequences on the program $P^{\Gamma_s(M)}$, the fact $\overline{p}$ is never reached. Therefore, the rules with $\overline{p}$ in the body are never used, and so, the program $P'^{\Gamma_s(M)}$ has the same least model: $\Gamma\Gamma_s(M) = \Gamma'\Gamma_s(M) = \Gamma'\Gamma'_s(M)$. The proof for (b) is completely analogous.

$\mapsto$ R

First observe that, for any M with $\overline{p} \not\in M$, $P^M = P'^M$ and $P_s^M = P_s'^M$, and so, $\Gamma(M) = \Gamma'(M)$ and $\Gamma_s(M) = \Gamma'_s(M)$. Now, if $M = \Gamma\Gamma_s(M)$ we have (as in the two previous proofs): $p \in M$ and $\overline{p} \in M$. Therefore, we immediately have $\Gamma_s(M) = \Gamma'_s(M)$ (the first part of (a)). Now, note that $\overline{p} \not\in \Gamma_s(M)$, because all the rules with $\overline{p}$ as head contain $not\ p$ in their bodies, and we had that $p \in M$.

By our first observation, this means that the reduct for P and P' w.r.t. $\Gamma_s(M)$ is the same one: $\Gamma\Gamma_s(M) = \Gamma'\Gamma_s(M) = \Gamma'\Gamma'_s(M)$. The proof for (b) is completely analogous.

Proof. **(theorem 2)** Simply note that the WFM in WFSX is defined as the three-valued interpretation $W = (W^+, W^-)$ with $W^+ = lfp(\Gamma\Gamma_s)$ and $W^- = \Gamma_s(M^+)$. As we have proved in lemma 1, any fixpoint of $\Gamma\Gamma_s$ is fixpoint of $\Gamma'\Gamma'_s$ and vice versa. So $W^+ = lfp(\Gamma'\Gamma'_s)$. Besides, as also proved in lemma 1, for any fixpoint M, $\Gamma_s(M) = \Gamma'_s(M)$. So $W^- = \Gamma'_s(W^+)$. Therefore, if W is WFM of P, it is WFM of P'. Finally, if P has no WFM, as the fixpoints for $\Gamma\Gamma_s$ and $\Gamma'\Gamma'_s$ coincide, then P' has no WFM.

Proof. **(theorem 3)** It follows from the previous results. Let $W = (W^+, W^-)$ and $X = (X^+, X^-)$. We consider (i) first. It is easy to see that any program P' containing the facts p and $\overline{p}$ is contradictory (has no fixpoints) in WFSX. As the transformations for WFS are also sound in WFSX, whenever we get the facts p and $\overline{p}$ in some of the transformed programs P', its WFM in WFSX is not defined, and so, the WFM for the original program is not defined as well. To prove (ii), it suffices with additionally applying lemma 1 for the resulting program P' after exhaustively applying all the WFS transformations. The facts of P' (i.e. W^+) are included in X^+ whereas the "non-head" atoms (i.e. W^-) are included in X^-. So $W \leq_F X$ for that program, and also for the original one.

Proof. **(theorem 4)** Let (U^+, U^-) be the trivial interpretation of P. We have to prove that:

i) $U^+ \stackrel{\text{def}}{=} facts(P) = lfp(\Gamma\Gamma_s)$
ii) $U^- \stackrel{\text{def}}{=} (\mathcal{H} - heads(P)) \cup \overline{facts(P)} = \mathcal{H} - \Gamma_s(U^+)$

We begin proving (ii). Consider $\Gamma_s(U^+)$, and more concretely, the reduct $P_s^{U^+}$. By non-applicability of $\vdash \xrightarrow{\mathtt{N}}$ program P cannot contain a rule with *not* p in the body, being p a fact of P. However, in P_s, any rule with $\overline{p}$ in the head contains *not* p in its body. So, $P_s^{U^+}$ is the result of deleting in P_s any rule whose head is in $\overline{facts(P)}$ plus the remaining default literals. As a first consequence $\Gamma_s(U^+) \cap \overline{facts(P)} = \emptyset$. But also, it is easy to see that $P_s^{U^+} \subseteq P^{\emptyset}$. By non-applicability of $\vdash \xrightarrow{\mathtt{L}}$ all the positive literals of P are included in $\Gamma(\emptyset)$ whereas by non-applicability of $\vdash \xrightarrow{\mathtt{C}}$ there is no positive literal of P in $\overline{facts(P)}$. As a result, all the rule bodies in $P^{\emptyset}$ are true w.r.t. $\Gamma(\emptyset)$ and so $\Gamma(\emptyset) = heads(P^{\emptyset}) = heads(P)$. Finally, since the rules $P^{\emptyset} - P_s^{U^+}$ are those with heads in $\overline{facts(P)}$ and these in their turn never occur in the bodies of P, we get that $\Gamma_s(U^+) = \Gamma(\emptyset) - \overline{facts(P)} = heads(P) - \overline{facts(P)}$. Then, it directly follows that $\mathcal{H} - \Gamma_s(U^+) = \mathcal{H} - (heads(P) - \overline{facts(P)}) = U^-$. Now, we proceed to prove (i). By lemma 1, the trivial interpretation (U^+, U^-) has less information than the WFM. So, $U^+ \subseteq lfp(\Gamma\Gamma_s)$ and it will suffice with showing that U^+ is simply a fixpoint: $\Gamma\Gamma_s(U^+) = U^+$. By (ii), $\Gamma\Gamma_s(U^+) = \Gamma(heads(P) - \overline{facts(P)})$. If we call $J = heads(P) - \overline{facts(P)}$, we want to establish the least model of P^J. By

non-applicability of $\overset{\texttt{P}}{\longmapsto}$and $\overset{\texttt{R}}{\longmapsto}$, given any $not\ p$ in P, $p \in heads(P) - \overline{facts(P)}$. So, all the rules with default literals are deleted in P^J. Now, by non-applicability of $\overset{\texttt{S}}{\longmapsto}$in P, any body atom P^J cannot belong to $facts(P) = facts(P^J)$. This means that, when computing $T_{P^J} \uparrow (\emptyset)$, rules with nonempty body are never used. In other words, the least model of P^J, is $facts(P') = facts(P)$. That is, $\Gamma(J) = \Gamma\Gamma_s(U^+) = facts(P) = U^+$.

Embedding Defeasible Logic into Logic Programs

Grigoris Antoniou[1] and Michael J. Maher[2]

[1] Department of Computer Science, University of Bremen
ga@tzi.de
[2] Department of Mathematical and Computer Sciences, Loyola University Chicago
mjm@math.luc.edu

Abstract. Defeasible reasoning is a simple but efficient approach to nonmonotonic reasoning that has recently attracted considerable interest and that has found various applications. Defeasible logic and its variants are an important family of defeasible reasoning methods. So far no relationship has been established between defeasible logic and mainstream nonmonotonic reasoning approaches.
In this paper we establish close links to known semantics of extended logic programs. In particular, we give a translation of a defeasible theory D into a program $P(D)$. We show that under a condition of decisiveness, the defeasible consequences of D correspond exactly to the sceptical conclusions of $P(D)$ under the stable model semantics. Without decisiveness, the result holds only in one direction (all defeasible consequences of D are included in all stable models of $P(D)$). If we wish a complete embedding for the general case, we need to use the Kunen semantics of $P(D)$, instead.

1 Introduction

Defeasible reasoning is a nonmonotonic reasoning [18] approach in which the gaps due to incomplete information are closed through the use of defeasible rules that are usually appropriate. Defeasible logics were introduced and developed by Nute over several years [20]. These logics perform defeasible reasoning, where a conclusion supported by a rule might be overturned by the effect of another rule. Roughly, a proposition p can be defeasibly proved ($+\partial p$) only when a rule supports it, and it has been demonstrated that no applicable rule supports $\neg p$; this demonstration makes use of statements $-\partial q$ which mean intuitively that an attempt to prove q defeasibly has failed finitely. These logics also have a monotonic reasoning component, and a priority on rules. One advantage of Nute's design was that it was aimed at supporting efficient reasoning, and in our work we follow that philosophy.

Defeasible reasoning has recently attracted considerable interest. Its use in various application domains has been advocated, including the modelling of regulations and business rules [19,12,1], modelling of contracts [22], legal reasoning [21] and agent negotiations [10]. In fact, defeasible reasoning (in the form of courteous logic programs [11]) provides a foundation for IBM's Business Rules

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 393–404, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Markup Language and for current W3C activities on rules. Therefore defeasible reasoning is arguably the most successful subarea in nonmonotonic reasoning as far as applications and integration to mainstram IT is concerned.

Recent theoretical work on defeasible logics has: (i) established some relationships to other logic programming approaches without negation as failure [2]; (ii) analysed the formal properties of these logics [4,14,15], and (iii) has delivered efficient implementations [17].

However the problem remains that defeasible logic is not firmly linked to the mainstream of nonmonotonic reasoning, in particular the semantics of logic programs. This paper aims at resolving this problem. Our initial approach is to consider answer set semantics of logic programs [9] and use a natural, direct translation (defeasible rules translated into "normal defaults"). We discuss why this translation cannot be successful. Then we define a second translation which makes use of control literals, similar to those used in [7].

Under this translation of a defeasible theory D into a logic program $P(D)$ we can show that

p is defeasibly provable in $D \iff$
p is included in all stable models of $P(D)$. $(*)$

However this result can only be shown under the additional condition of *decisiveness*: for every literal q, either $+\partial q$ or $-\partial q$ can be derived. A sufficient condition for decisiveness is the absence of cycles in the atom dependency graph.

If we wish to drop decisiveness, $(*)$ holds only in one direction, from left to right. We show that if we wish the equivalence in the general case, we need to use another semantics for logic programs, namely Kunen semantics [13].

There is previous work relating defeasible logic and logic programs. [16] showed that the notion of failure in defeasible logic corresponds to Kunen semantics. That work used a metaprogram to express defeasible logic in logic programming terms. The translation we present here is more direct. [6] provided a translation of a different defeasible logic to logic programs with well-founded semantics, but that translation does not provide a characterization of the defeasible logic.

The paper is organised as follows. Sections 2 and 3 present the basics of defeasible logic and logic programming semantics, respectively. Section 4 presents our translation and its ideas, while section 5 contains the main results.

2 Defeasible Logic

2.1 A Language for Defeasible Reasoning

A defeasible theory (a knowledge base in defeasible logic) consists of three different kinds of knowledge: strict rules, defeasible rules, and a superiority relation. (Fuller versions of defeasible logic also have facts and defeaters, but [4] shows that they can be simulated by the other ingredients).

Strict rules are rules in the classical sense: whenever the premises are indisputable (e.g. facts) then so is the conclusion. An example of a strict rule is "Emus are birds". Written formally:

$$emu(X) \rightarrow bird(X).$$

Defeasible rules are rules that can be defeated by contrary evidence. An example of such a rule is "Birds typically fly"; written formally:

$$bird(X) \Rightarrow flies(X).$$

The idea is that if we know that something is a bird, then we may conclude that it flies, *unless there is other, not inferior, evidence suggesting that it may not fly.*

The *superiority relation* among rules is used to define priorities among rules, that is, where one rule may override the conclusion of another rule. For example, given the defeasible rules

$$\begin{array}{ll} r: & bird(X) \Rightarrow flies(X) \\ r': & brokenWing(X) \Rightarrow \neg flies(X) \end{array}$$

which contradict one another, no conclusive decision can be made about whether a bird with broken wings can fly. But if we introduce a superiority relation $>$ with $r' > r$, with the intended meaning that r' is strictly stronger than r, then we can indeed conclude that the bird cannot fly.

It is worth noting that, in defeasible logic, priorities are *local* in the following sense: Two rules are considered to be competing with one another only if they have complementary heads. Thus, since the superiority relation is used to resolve conflicts among competing rules, it is only used to compare rules with complementary heads; the information $r > r'$ for rules r, r' without complementary heads may be part of the superiority relation, but has no effect on the proof theory.

[4] showed that there is a constructive, conclusion-preserving transformation which takes an arbitrary defeasible theory and translates it into a theory which has only strict rules and defeasible rules. For the sake of simplicity, we will assume in this paper that indeed a defeasible theory consists only of strict rules and defeasible rules.

2.2 Formal Definition

In this paper we restrict attention to essentially propositional defeasible logic. Rules with free variables are interpreted as rule schemas, that is, as the set of all ground instances; in such cases we assume that the Herbrand universe is finite. We assume that the reader is familiar with the notation and basic notions of propositional logic. If q is a literal, $\sim q$ denotes the complementary literal (if q is a positive literal p then $\sim q$ is $\neg p$; and if q is $\neg p$, then $\sim q$ is p).

Rules are defined over a *language* (or *signature*) Σ, the set of propositions (atoms) and labels that may be used in the rule.

A *rule* $r : A(r) \hookrightarrow C(r)$ consists of its unique *label* r, its *antecedent* $A(r)$ ($A(r)$ may be omitted if it is the empty set) which is a finite set of literals, an arrow $\hookrightarrow$ (which is a placeholder for concrete arrows to be introduced in a moment), and its *head* (or *consequent*) $C(r)$ which is a literal. In writing rules often we omit set notation for antecedents and sometimes we omit the label when it is not relevant for the context. There are two kinds of rules, each represented by a different arrow. Strict rules use $\rightarrow$ and defeasible rules use $\Rightarrow$.

Given a set R of rules, we denote the set of all strict rules in R by R_s, and the set of defeasible rules in R by R_d. $R[q]$ denotes the set of rules in R with consequent q.

A *defeasible theory* D is a finite set of rules R.

2.3 Proof Theory

A *conclusion* of a defeasible theory D is a tagged literal. A conclusion has one of the following four forms:

- $+\Delta q$, which is intended to mean that the literal q is definitely provable, using only strict rules.
- $-\Delta q$, which is intended to mean that q is provably not strictly provable (finite failure).
- $+\partial q$, which is intended to mean that q is defeasibly provable in D.
- $-\partial q$ which is intended to mean that we have proved that q is not defeasibly provable in D.

Provability is defined below. It is based on the concept of a *derivation* (or *proof*) in $D = R$. A derivation is a finite sequence $P = P(1), \ldots, P(n)$ of tagged literals satisfying the following conditions. The conditions are essentially inference rules phrased as conditions on proofs. $P(1..i)$ denotes the initial part of the sequence P of length i.

$+\Delta$: If $P(i+1) = +\Delta q$ then
$\exists r \in R_s[q]\ \forall a \in A(r) : +\Delta a \in P(1..i)$

That means, to prove $+\Delta q$ we need to establish a proof for q using strict rules only. This is a deduction in the classical sense – no proofs for the negation of q need to be considered (in contrast to defeasible provability below, where opposing chains of reasoning must be taken into account, too).

$-\Delta$: If $P(i+1) = -\Delta q$ then
$\forall r \in R_s[q]\ \exists a \in A(r) : -\Delta a \in P(1..i)$

The definition of $-\Delta$ is the so-called *strong negation* of $+\Delta$: normal negation rules like De-Morgan rules are applied to the definition, $+$ is replaced by $-$, and vice versa. Therefore in the following we may omit giving inference conditions of both $+$ and $-$.

$+\partial$: If $P(i+1) = +\partial q$ then either
(1) $+\Delta q \in P(1..i)$ or
(2) (2.1) $\exists r \in R[q]\ \forall a \in A(r) : +\partial a \in P(1..i)$ and
(2.2) $-\Delta \sim q \in P(1..i)$ and
(2.3) $\forall s \in R[\sim q] \exists a \in A(s) : -\partial a \in P(1..i)$

Let us illustrate this definition. To show that q is provable defeasibly we have two choices: (1) We show that q is already definitely provable; or (2) we need to argue using the defeasible part of D as well. In particular, we require that there must be a strict or defeasible rule with head q which can be applied (2.1). But now we need to consider possible "counterattacks", that is, reasoning chains in support of $\sim q$. To be more specific: to prove q defeasibly we must show that $\sim q$ is not definitely provable (2.2). Also (2.3) we must consider the set of all rules which are not known to be inapplicable and which have head $\sim q$. Essentially each such rule s attacks the conclusion q. For q to be provable, each such rule s must have been established as non-applicable.

A defeasible theory D is called *decisive* iff for every literal p, either $D \vdash -\partial p$ or $D \vdash +\partial p$. Not every defeasible theory satisfies this property. For example, in the theory consisting of the single rule

$$p \Rightarrow p$$

neither $-\partial p$ nor $+\partial p$ is provable. However, decisiveness is guaranteed in defeasible theories with an acyclic atom dependency graph [5].

3 Semantics of Logic Programs

A *logic program* P is a finite set of program clauses. A *program clause* r has the form

$$A \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

where $A, B_1, \ldots B_n, C_1, \ldots, C_m$ are positive literals.

3.1 Stable Model Semantics

Let M be a subset of the Herbrand base. We call a ground program clause

$$A \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$$

irrelevant w.r.t. M if at least one C_i is included in M. Given a logic program P, we define the *reduct of P w.r.t. M*, denoted by P^M, to be the logic program obtained from $ground(P)$ by

1. removing all clauses that are irrelevant w.r.t. M, and
2. removing all premises *not* C_i from all remaining program clauses.

Note that the reduct P^M is a definite logic program, and we are no longer faced with the problem of assigning semantics to negation, but can use the least Herbrand model instead.

M is a *stable model* of P iff $M = \mathcal{M}_{P^M}$.

3.2 Kunen Semantics

Kunen semantics [13] is a 3-valued semantics for logic programs. An *interpretation* is a mapping from ground atoms to one of the three truth values **t**, **f** and **u**, which denote true, false and unknown, respectively. This mapping can be extended to arbitrary formulas using Kleene's 3-valued logic.

Kleene's truth tables can be summarized as follows. If φ is a boolean combination of atoms with truth values **t**, **f** or **u**, its truth value is **t** iff all possible ways of putting **t** or **f** for the various **u**-values lead to a value **t** being computed in ordinary (2-valued) logic; φ gets the value **f** iff *not* φ gets the value **t**; and φ gets the value **u** otherwise. These truth values can be extended in the obvious way to predicate logic, thinking of the quantifiers as infinite conjunctions or disjunctions.

The Kunen semantics of a program P is obtained from a sequence $\{I_n\}$ of interpretations, defined as follows:

1. $I_0(\alpha) = \mathbf{u}$ for every atom α.
2. $I_{n+1}(\alpha) = \mathbf{t}$ iff for some clause $\alpha \leftarrow \varphi$ in the program, $I_n(\varphi) = \mathbf{t}$.
3. $I_{n+1}(\alpha) = \mathbf{f}$ iff for all clauses $\alpha \leftarrow \varphi$ in the program, $I_n(\varphi) = \mathbf{f}$.
4. $I_{n+1}(\alpha) = \mathbf{u}$ if neither 2. nor 3. applies.

We shall say that the Kunen semantics of P supports α, written $P \vdash_K \alpha$, iff there is an interpretation I_n, for some finite n, such that $I_n(\alpha) = \mathbf{t}$.

4 A Translation of Defeasible Theories into Logic Programs

4.1 A Direct Translation that Fails

Here we consider the most natural translation of a defeasible theory into logic programs. Since in defeasible logic both positive and negative literals are used, the translation in this section yields an extended logic program. We will consider the answer set semantics for extended logic programs [9], which is a generalisation of the stable model semantics.

A natural translation of a defeasible theory into a logic program would look as follows. A strict rule

$$\{q_1, \ldots, q_n\} \rightarrow p$$

is translated into the program clause

$$p \leftarrow q_1, \ldots, q_n.$$

And a defeasible rule

$$\{q_1, \ldots, q_n\} \Rightarrow p$$

is translated into

$p \leftarrow q_1, \ldots, q_n, not \sim p.$

Unfortunately this translation does not lead to a correspondence between the defeasible conclusions and the sceptical conclusions in answer set semantics, as the following example demonstrates.

Example 1. Consider the defeasible theory

$\Rightarrow p$
$\Rightarrow \neg p$
$\Rightarrow q$
$p \Rightarrow \neg q$

Here q is defeasibly provable because the only rule with head $\neg q$ is not applicable, because $-\partial p$. However, the translated logic program

$p \leftarrow not\ \neg p.$
$\neg p \leftarrow not\ p.$
$q \leftarrow not\ \neg q.$
$\neg q \leftarrow p, not\ q.$

has three answer sets, $\{p, q\}$, $\{p, \neg q\}$ and $\{\neg p, q\}$. Thus none of $p, \neg p, q, \neg q$ is included in all stable models.

The example above demonstrates that the translation does not capture the ambiguity blocking behaviour of defeasible logic (ambiguity of p is not propagated to the dependent atom q). But even if we try to overcome this problem by considering an ambiguity propagating defeasible logic instead [3], there remains the problem of floating conclusions, as the following example demonstrates.

Example 2. Consider the defeasible theory

$\Rightarrow p$
$\Rightarrow \neg p$
$p \Rightarrow q$
$\neg p \Rightarrow q$

In defeasible logic, q is not defeasibly provable because neither p nor $\neg p$ are defeasibly provable. However, the translation

$p \leftarrow not\ \neg p.$
$\neg p \leftarrow not\ p.$
$q \leftarrow p, not\ \neg q.$
$q \leftarrow \neg p, not\ \neg q.$

has two answer sets, $\{p, q\}$ and $\{\neg p, q\}$, so q is a sceptical conclusion under the answer set semantics.

Finally, there is a flaw in the use of explicit (or classical) negation in the translated program to represent explicit negation in the defeasible theory. Logic programs, under the answer set semantics, react to an inconsistency by inferring all literals whereas defeasible logic is paraconsistent. As a consequence, the translated program does not reflect the behavior of defeasible logic when an inconsistency is involved, as in the following example.

Example 3. Consider the defeasible theory

$\rightarrow p$
$\rightarrow \neg p$
$\rightarrow q$

The translation is

$p \leftarrow$
$\neg p \leftarrow$
$q \leftarrow$

The only answer set of this program is $\{p, \neg p, q, \neg q\}$ which does not agree with defeasible logic: the literal $\neg q$ is included in the answer set but is not strictly provable in defeasible logic.

4.2 A Translation Using Control Literals

Above we outlined the reasons why a direct translation of a defeasible theory into a logic program must fail. Here we propose a different translation which uses "control literals" that carry meaning regarding the applicability status of rules.

First we translate strict rules. In defeasible logic, strict rules play a twofold role: on one hand they can be used to derive undisputed conclusions if all their antecedents have been strictly proved. And on the other hand they can be used essentially as defeasible rules, if their antecedents are defeasibly provable. These two roles can be clearly seen in the inference condition $+\partial$ is section 2.

To capture both uses we introduce mutually disjoint copies *strict-p* and def-p, for all literals p. Note that this way the logic program we get does not have classical negation, as in the previous section. Among others, this solution avoids the problem illustrated by Example 3.

Given a strict rule

$r : \{q_1, \ldots, q_n\} \rightarrow p$

we translate it into the program clause

$a(r) : \textit{strict-}p \leftarrow \textit{strict-}q_1, \ldots, \textit{strict-}q_n.$

Additionally, we introduce the clause

$b(p) : def\text{-}p \leftarrow strict\text{-}p$

for every literal p. Intuitively, *strict-p* means that p is strictly provable, and *def-p* that p is defeasibly provable. And the clause $b(p)$ corresponds to the condition (1) in the $+\partial$ inference condition: a literal p is defeasibly provable if it is strictly provable.

Next we turn our attention to defeasible rules and consider

$r : \{q_1, \ldots, q_n\} \Rightarrow p$

r is translated into the following set of clauses:

$d_1(r) : def\text{-}p \leftarrow def\text{-}q_1, \ldots, def\text{-}q_n, not\ strict\text{-}{\sim} p, ok(r).$
$d_2(r) : ok(r) \leftarrow ok'(r, s_1), \ldots, ok'(r, s_m)$, where $R[{\sim}p] = \{s_1, \ldots, s_m\}$.
$d_3(r, s) : ok'(r, s) \leftarrow blocked(s)$, for all $s \in R[{\sim}p]$.
$d_4(r, q_i) : blocked(r) \leftarrow not\ def\text{-}q_i$, for all $i \in \{1, \ldots, n\}$.

In the above, the predicates ok, ok' and *blocked* are new and pairwise disjoint.

- $d_1(r)$ says that to prove p defeasibly by applying r, we must prove all the antecedents of r, the negation of p should not be strictly provable, and it must be ok to apply r.
- The clause $d_2(r)$ says when it is ok to apply a rule r with head p: we must check that it is ok to apply r w.r.t. every rule with head ${\sim}p$.
- $d_3(r, s)$ says that it is ok to apply r w.r.t. s if s is blocked. Obviously this clause would look more complicated if we had considered priorities, instead of compiling them into the defeasible theory prior to the translation. Indeed, in the present framework we could have used a somewhat simpler translation, replacing d_1, d_2, and d_3 by

  $$def\text{-}p \leftarrow def\text{-}q_1, \ldots, def\text{-}q_n, not\ strict\text{-}{\sim} p, blocked(s_1), \ldots, blocked(s_m)$$

  but chose to maintain the intuitive nature of the translation in its present form.
- Finally, d_4 specifies the only way a rule r can be blocked: it must be impossible to prove one of its antecedents.

For a defeasible theory D we define $P(D)$ to be the union of all clauses $a(r), b(p), d_1(r), d_2(r), d_3(r, s)$ and $d_4(r, q_i)$.

Example 4. We consider the defeasible theory from Example 1:

$r_1 : \ \Rightarrow p$
$r_2 : \ \Rightarrow \neg p$
$r_3 : \ \Rightarrow q$
$r_4 : \ p \Rightarrow \neg q$

Its translation looks as follows:

$d_1(r_1):\ def\text{-}p \leftarrow not\ strict\text{-}\neg p, ok(r_1).$
$d_2(r_1):\ ok(r_1) \leftarrow ok'(r_1, r_2).$
$d_3(r_1):\ ok'(r_1, r_2) \leftarrow blocked(r_2).$

$d_1(r_2):\ def\text{-}\neg p \leftarrow not\ strict\text{-}p, ok(r_2).$
$d_2(r_2):\ ok(r_2) \leftarrow ok'(r_2, r_1).$
$d_3(r_2):\ ok'(r_2, r_1) \leftarrow blocked(r_1).$

$d_1(r_3):\ def\text{-}q \leftarrow not\ strict\text{-}\neg q, ok(r_3).$
$d_2(r_3):\ ok(r_3) \leftarrow ok'(r_3, r_4).$
$d_3(r_3):\ ok'(r_3, r_4) \leftarrow blocked(r_4).$

$d_1(r_4):\ def\text{-}\neg q \leftarrow def\text{-}p, not\ strict\text{-}q, ok(r_4).$
$d_2(r_4):\ ok(r_4) \leftarrow ok'(r_4, r_3).$
$d_3(r_4):\ ok'(r_4, r_3) \leftarrow blocked(r_3).$
$d_4(r_4):\ blocked(r_4) \leftarrow not\ def\text{-}p.$

$\{blocked(r_4), ok'(r_3, r_4), ok(r_3), def\text{-}q\}$ is the only stable model.

5 Properties of the Translation

We begin with an observation on the size of the translation. By the size of a defeasible theory, we mean the number of rules.

Proposition 1. *The size of $P(D)$ is bound by $L + n \times (3 + L) + n^2$, where n is the number of rules in D and L the number of literals occurring in D.*

Next we establish relationships between D and its translation $P(D)$. To do so we must select appropriate logic program semantics to interpret *not*. First we consider stable model semantics.

Theorem 1.

(a) $D \vdash +\Delta p \Leftrightarrow$ strict-p is included in all stable models of $P(D)$.
(b) $D \vdash -\Delta p \Rightarrow$ strict-p is not included in any stable model of $P(D)$.
(c) If D is decisive on definite conclusions then the implication (b) is also true in the opposite direction.

A defeasible theory D is *decisive on definite conclusions* if, for every literal p, either $D \vdash +\Delta p$ or $D \vdash -\Delta p$.

Theorem 2.

(a) $D \vdash +\partial p \Rightarrow def\text{-}p$ is included in all stable models of $P(D)$.
(b) $D \vdash -\partial p \Rightarrow def\text{-}p$ is not included in any stable model of $P(D)$.
(c) If D is decisive then the implications (a) and (b) are also true in the opposite direction.

That is, if D is decisive, then the stable model semantics of $P(D)$ corresponds to the provability in defeasible logic. However part *(c)* is not true in the general case, as the following example shows.

Example 5. Consider the defeasible theory

$r_1 : \Rightarrow \neg p$
$r_2 : p \Rightarrow p$

In defeasible logic, $+\partial\neg p$ cannot be proven because we cannot derive $-\partial p$. However, $blocked(r_2)$ is included in the only stable model of $P(D)$, so $def\text{-}\neg p$ is a sceptical conclusion of $P(D)$ under stable model semantics.

If we wish to have an equivalence result without the condition of decisiveness, then we must use a different logic programming semantics, namely Kunen semantics.

Theorem 3.

(a) $D \vdash +\Delta p \Leftrightarrow P(D) \vdash_K strict\text{-}p.$
(b) $D \vdash -\Delta p \Leftrightarrow P(D) \vdash_K not\ strict\text{-}p.$
(c) $D \vdash +\partial p \Leftrightarrow P(D) \vdash_K def\text{-}p.$
(d) $D \vdash -\partial p \Leftrightarrow P(D) \vdash_K not\ def\text{-}p.$

6 Conclusion

We motivated and presented a translation of defeasible theories into logic programs, such that the defeasible conclusions of the former correspond exactly with the sceptical conclusions of the latter under the stable model semantics, if a condition of decisiveness is satisfied. If decisiveness is not satisfied, we have to use Kunen semantics instead.

This paper closes an important gap in the theory of nonmonotonic reasoning, in that it relates defeasible logic with mainstream semantics of logic programming. This result is particularly important, since defeasible reasoning is one of the most successful nonmonotonic reasoning paradigms in applications.

References

1. G. Antoniou, D. Billington and M. J. Maher. On the analysis of regulations using defeasible rules. In *Proc. 32nd Hawaii International Conference on Systems Science*, 1999. 393
2. G. Antoniou, M. J. Maher and D. Billington. Defeasible Logic versus Logic Programming without Negation as Failure, Journal of Logic Programming, 42 (2000): 47-57. 394
3. G. Antoniou, D. Billington, G. Governatori and M. J. Maher. A flexible framework for defeasible logics. In *Proc. 17th American National Conference on Artificial Intelligence (AAAI-2000)*, 405-410. 399

4. G. Antoniou, D. Billington, G. Governatori and M. J. Maher. Representation results for defeasible logic. *ACM Transactions on Computational Logic* 2 (2001): 255–287 394, 395
5. D. Billington. Defeasible Logic is Stable. *Journal of Logic and Computation* 3 (1993): 370–400. 397
6. G. Brewka. On the Relationship between Defeasible Logic and Well-Founded Semantics. In *Proc. Logic Programming and Nonmonotonic Reasoning Conference*, LNCS 2173, 2001, 121–132. 394
7. J. P. Delgrande, T Schaub and H. Tompits. Logic Programs with Compiled Preferences. In *Proc. ECAI'2000*, 464–468. 394
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. International Conference on Logic Programming*, MIT Press 1988, 1070–1080.
9. M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing* 9 (1991): 365–385. 394, 398
10. G. Governatori, A. ter Hofstede and P. Oaks. Defeasible Logic for Automated Negotiation. In *Proc. Fifth CollECTeR Conference on Electronic Commerce*, Brisbane 2000. 393
11. B. N. Grosof. Prioritized conflict handling for logic programs. In *Proc. International Logic Programming Symposium*, MIT Press 1997, 197–211. 393
12. B. N. Grosof, Y. Labrou and H. Y. Chan. A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In *Proc. 1st ACM Conference on Electronic Commerce (EC-99)*, ACM Press 1999. 393
13. K. Kunen. Negation in Logic Programming. *Journal of Logic Programming* 4 (1987): 289–308. 394, 398
14. M. J. Maher. A Denotational Semantics for Defeasible Logic. In *Proc. First International Conference on Computational Logic*, LNAI 1861, Springer, 2000, 209-222. 394
15. M. J. Maher. Propositional Defeasible Logic has Linear Complexity. *Theory and Practice of Logic Programming*, 1 (6), 691–711, 2001. 394
16. M. Maher and G. Governatori. A Semantic Decomposition of Defeasible Logics. In *Proc. American National Conference on Artificial Intelligence (AAAI-99)*, AAAI/MIT Press 1999, 299–305. 394
17. M. J. Maher, A. Rock, G. Antoniou, D. Billington and T. Miller. Efficient Defeasible Reasoning Systems. In *Proc. 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2000)*, IEEE 2000, 384-392. 394
18. V. Marek and M. Truszczynski. *Nonmonotonic Logic*. Springer 1993. 393
19. L. Morgenstern. Inheritance Comes of Age: Applying Nonmonotonic Techniques to Problems in Industry. *Artificial Intelligence*, 103 (1998): 1–34. 393
20. D. Nute. Defeasible Logic. In D. M. Gabbay, C. J. Hogger and J. A. Robinson (eds.): *Handbook of Logic in Artificial Intelligence and Logic Programming Vol. 3*, Oxford University Press 1994, 353–395. 393
21. H. Prakken. *Logical Tools for Modelling Legal Argument: A Study of Defeasible Reasoning in Law*. Kluwer Academic Publishers 1997. 393
22. D. M. Reeves, B. N. Grosof, M. P. Wellman, and H. Y. Chan. Towards a Declarative Language for Negotiating Executable Contracts, *Proceedings of the AAAI-99 Workshop on Artificial Intelligence in Electronic Commerce (AIEC-99)*, AAAI Press / MIT Press, 1999. 393

A Polynomial Translation of Logic Programs with Nested Expressions into Disjunctive Logic Programs: Preliminary Report

David Pearce[1], Vladimir Sarsakov[2], Torsten Schaub[2⋆],
Hans Tompits[3], and Stefan Woltran[3]

[1] European Commission, DG Information Society – F1
BU33 3/58, Rue de le Loi 200, B-1049 Brussels
David.Pearce@cec.eu.int
[2] Institut für Informatik, Universität Potsdam
Postfach 60 15 53, D–14415 Potsdam Germany
sarsakov@rz.uni-potsdam.de
torsten@cs.uni-potsdam.de
[3] Institut für Informationssysteme 184/3, Technische Universität Wien
Favoritenstraße 9–11, A–1040 Wien, Austria
{tompits,stefan}@kr.tuwien.ac.at

Abstract. Nested logic programs have recently been introduced in order to allow for arbitrarily nested formulas in the heads and the bodies of logic program rules under the answer sets semantics. Previous results show that nested logic programs can be transformed into standard (unnested) disjunctive logic programs in an elementary way, applying the negation-as-failure operator to body literals only. This is of great practical relevance since it allows us to evaluate nested logic programs by means of off-the-shelf disjunctive logic programming systems, like DLV. However, it turns out that this straightforward transformation results in an exponential blow-up in the worst-case, despite the fact that complexity results indicate that there is a polynomial translation among both formalisms. In this paper, we take up this challenge and provide a polynomial translation of logic programs with nested expressions into disjunctive logic programs. Moreover, we show that this translation is modular and (strongly) faithful. We have implemented both the straightforward as well as our advanced transformation; the resulting compiler serves as a front-end to DLV and is publicly available on the Web.

1 Introduction

Lifschitz, Tang, and Turner [24] recently extended the answer set semantics [12] to a class of logic programs in which arbitrarily nested formulas, formed from literals using negation as failure, conjunction, and disjunction, constitute the heads and bodies of rules. These so-called *nested logic programs* generalise the

⋆ Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 405–420, 2002.
© Springer-Verlag Berlin Heidelberg 2002

well-known classes of *normal*, *generalised*, *extended*, and *disjunctive logic programs*, respectively. Despite their syntactically much more restricted format, the latter classes are well recognised as important tools for knowledge representation and reasoning. This is reflected by the fact that several practically relevant applications have been developed recently using these types of programs (cf., e.g., [22, 3, 11, 16]), which in turn is largely fostered by the availability of efficient solvers for the answer set semantics, most notably `DLV` [8, 9] and `Smodels` [27].

In this paper, we are interested in utilising these highly performant solvers for interpreting nested logic programs. We address this problem by providing a translation of nested logic programs into disjunctive logic programs. In contrast to previous work, our translation is guaranteed to be polynomial in time and space, as suggested by related complexity results [32]. More specifically, we provide a translation, σ, from nested logic programs into disjunctive logic programs possessing the following properties:

- σ maps nested logic programs over an alphabet $\mathcal{A}_1$ into disjunctive logic programs over an alphabet $\mathcal{A}_2$, where $\mathcal{A}_1 \subseteq \mathcal{A}_2$;
- the size of $\sigma(\Pi)$ is polynomial in the size of Π;
- σ is *faithful*, i.e., for each program Π over alphabet $\mathcal{A}_1$, there is a one-to-one correspondence between the answer sets of Π and sets of form $I \cap \mathcal{A}_1$, where I is an answer set of $\sigma(\Pi)$; and
- σ is *modular*, i.e., $\sigma(\Pi \cup \Pi') = \sigma(\Pi) \cup \sigma(\Pi')$, for each program Π, Π'.

Moreover, we have implemented translation σ, serving as a front-end for the logic programming system `DLV`.

The construction of σ relies on the introduction of new *labels*, abbreviating subformula occurrences. This technique is derived from *structure-preserving normal form translations* [36, 33], frequently employed in the context of automated deduction (cf. [1] for an overview). We use here a method adapted from a structure-preserving translation for intuitionistic logic as described in [26].

Regarding the faithfulness of σ, we actually provide a somewhat stronger condition, referred to as *strong faithfulness*, expressing that, for any programs Π and Π' over alphabet $\mathcal{A}_1$, there is a one-to-one correspondence between the answer sets of $\Pi \cup \Pi'$ and sets of form $I \cap \mathcal{A}_1$, where I is an answer set of $\sigma(\Pi) \cup \Pi'$. This condition means that we can add to a given program Π *any nested program* Π' and still recover the answer sets of the combined program $\Pi \cup \Pi'$ from $\sigma(\Pi) \cup \Pi'$; in particular, for any nested logic program Π, we may choose to translate, in a semantics-preserving way, only an arbitrary *program part* $\Pi_0 \subseteq \Pi$ and leave the remaining part $\Pi \setminus \Pi_0$ unchanged. For instance, if Π_0 is already a disjunctive logic program, we do not need to translate it again into another (equivalent) disjunctive logic program. Strong faithfulness is closely related to the concept of *strong equivalence* [23] (see below).

In order to have a sufficiently general setting for our purposes, we base our investigation on *equilibrium logic* [28], a generalisation of the answer set semantics for nested logic programs. Equilibrium logic is a form of minimal-model reasoning in the *logic of here-and-there*, which is intermediate between classical

logic and intuitionistic logic (the logic of here-and-there is also known as *Gödel's three-valued logic* in view of [14]). As shown in [28, 29, 23], logic programs can be viewed as a special class of formulas in the logic of here-and-there such that, for each program Π, the answer sets of Π are given by the equilibrium models of Π, where the latter Π is viewed as a set of formulas in the logic of here-and-there.

The problem of implementing nested logic programs has already been addressed in [32], where (linear-time constructible) encodings of the basic reasoning tasks associated with this language into quantified Boolean formulas are described. These encodings provide a straightforward implementation for nested logic programs by appeal to off-the-shelf solvers for quantified Boolean formulas (like, e.g., the systems proposed in [4, 10, 13, 20, 21, 34]). Besides the encodings into quantified Boolean formulas, a further result of [32] is that nested logic programs possess the same worst-case complexity as disjunctive logic programs, i.e., the main reasoning tasks associated with nested logic programs lie at the second level of the polynomial hierarchy. From this result it follows that nested logic programs can in turn be efficiently reduced to disjunctive logic programs. Hence, given such a reduction, solvers for the latter kinds of programs, like, e.g., `DLV` or `Smodels`, can be used to compute the answer sets of nested logic programs. The main goal of this paper is to construct a reduction of this type.

Although results by Lifschitz, Tang, and Turner [24] (together with transformation rules given in [19]) provide a method to translate nested logic programs into disjunctive ones, that approach suffers from the drawback of an exponential blow-up of the resulting disjunctive logic programs in the worst case. This is due to the fact that this translation relies on distributivity laws yielding an exponential increase of program size whenever the given program contains rules whose heads are in disjunctive normal form or whose bodies are in conjunctive normal form, and the respective expressions are not simple disjunctions or conjunctions of literals. Our translation, on the other hand, is always polynomial in the size of its input program.

Finally, we mention that structure-preserving normal form translations in the logic of here-and-there are also studied, yet in much more general settings, by Baaz and Fermüller [2] as well as by Hähnle [15]; there, whole classes of finite-valued Gödel logics are investigated. Unfortunately, these normal form translations are not suitable for our purposes, because they do not enjoy the particular form of programs required here.

2 Preliminaries

We deal with propositional languages and use the logical symbols $\top$, $\bot$, $\neg$, $\vee$, $\wedge$, and $\rightarrow$ to construct formulas in the standard way. We write $\mathcal{L}_{\mathcal{A}}$ to denote a language over an alphabet $\mathcal{A}$ of *propositional variables* or *atoms*. Formulas are denoted by Greek lower-case letters (possibly with subscripts). As usual, *literals* are formulas of form v or $\neg v$, where v is some variable or one of $\top, \bot$.

Besides the semantical concepts introduced below, we also make use of the semantics of classical propositional logic. By a (*classical*) *interpretation*, I, we

understand a set of variables. Informally, a variable v is true under I iff $v \in I$. The truth value of a formula ϕ under interpretation I, in the sense of classical propositional logic, is determined in the usual way.

2.1 Logic Programs

The central objects of our investigation are logic programs with nested expressions, introduced by Lifschitz *et al.* [24]. These kinds of programs generalise normal logic programs by allowing bodies and heads of rules to contain arbitrary Boolean formulas. For reasons of simplicity, we deal here only with languages containing one kind of negation, however, corresponding to default negation. The extension to the general case where strong negation is also permitted is straightforward and proceeds in the usual way.

We start with some basic notation. A formula whose sentential connectives comprise only $\wedge$, $\vee$, or $\neg$ is called an *expression*. A rule, r, is an ordered pair of form

$$H(r) \leftarrow B(r),$$

where $B(r)$ and $H(r)$ are expressions. $B(r)$ is called the *body* of r and $H(r)$ is the *head* of r. We say that r is a *generalised disjunctive rule* if $B(r)$ is a conjunction of literals and $H(r)$ is a disjunction of literals; r is a *disjunctive rule* iff it is a generalised disjunctive rule containing no negated atom in its head; finally, if r is a rule containing no negation at all, then r is called *basic*. A *nested logic program*, or simply a *program*, Π, is a finite set of rules. Π is a *generalised disjunctive logic program* iff it contains only generalised disjunctive rules. Likewise, Π is a *disjunctive logic program* iff Π contains only disjunctive rules, and Π is *basic* iff each rule in Π is basic. We say that Π is a program *over* alphabet $\mathcal{A}$ iff all atoms occurring in Π are from $\mathcal{A}$. The set of all atoms occurring in program Π is denoted by $var(\Pi)$. We use $NLP_{\mathcal{A}}$ to denote the class of all nested logic programs over alphabet $\mathcal{A}$; furthermore, $DLP_{\mathcal{A}}$ stands for the subclass of $NLP_{\mathcal{A}}$ containing all disjunctive logic programs over $\mathcal{A}$; and $GDLP_{\mathcal{A}}$ is the class of all generalised disjunctive logic programs over $\mathcal{A}$. Further classes of programs are introduced in Section 4.

In what follows, we associate to each rule r a corresponding formula $\hat{r} = B(r) \rightarrow H(r)$ and, accordingly, to each program Π a corresponding set of formulas $\hat{\Pi} = \{\hat{r} \mid r \in \Pi\}$.

Let Π be a basic program over $\mathcal{A}$ and $I \subseteq \mathcal{A}$ a (classical) interpretation. We say that I is a *model* of Π iff it is a model of the associated set $\hat{\Pi}$ of formulas. Furthermore, given an (arbitrary) program Π over $\mathcal{A}$, the *reduct*, Π^I, of Π with respect to I is the basic program obtained from Π by replacing every occurrence of an expression $\neg\psi$ in Π which is not in the scope of any other negation by $\bot$ if ψ is true under I, and by $\top$ otherwise. I is an *answer set* (or *stable model*) of Π iff it is a minimal model (with respect to set inclusion) of the reduct Π^I. The collection of all answer sets of Π is denoted by $AS_{\mathcal{A}}(\Pi)$.

Two logic programs, Π_1 and Π_2, are *equivalent* iff they possess the same answer sets. Following Lifschitz *et al.* [23], we call Π_1 and Π_2 *strongly equivalent* iff, for every program Π, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ are equivalent.

2.2 Equilibrium Logic

Equilibrium logic is an approach to nonmonotonic reasoning that generalises the answer set semantics for logic programs. We use this particular formalism because it offers a convenient logical language for dealing with logic programs under the answer set semantics. It is defined in terms of the logic of here-and-there, which is intermediate between classical logic and intuitionistic logic. Equilibrium logic was introduced in [28] and further investigated in [29]; proof theoretic studies of the logic can be found in [31, 30].

Generally speaking, the logic of here-and-there is an important tool for analysing various properties of logic programs. For instance, as shown in [23], the problem of checking whether two logic programs are strongly equivalent can be expressed in terms of the logic of here-and-there (cf. Proposition 2 below).

The semantics of the logic of here-and-there is defined by means of two worlds, H and T, called "here" and "there". It is assumed that there is a total order, $\leq$, defined between these worlds such that $\leq$ is reflexive and $H \leq T$. As in ordinary Kripke semantics for intuitionistic logic, we can imagine that in each world a set of atoms is verified and that, once verified "here", an atom remains verified "there".

Formally, by an *HT-interpretation*, $\mathcal{I}$, we understand an ordered pair $\langle I_H, I_T \rangle$ of sets of atoms such that $I_H \subseteq I_T$. We say that $\mathcal{I}$ is an HT-interpretation *over* $\mathcal{A}$ if $I_T \subseteq \mathcal{A}$. The set of all HT-interpretations over $\mathcal{A}$ is denoted by $INT_{\mathcal{A}}$. An HT-interpretation $\langle I_H, I_T \rangle$ is *total* if $I_H = I_T$.

The truth value, $\nu_{\mathcal{I}}(w, \phi)$, of a formula ϕ at a world $w \in \{H, T\}$ in an HT-interpretation $\mathcal{I} = \langle I_H, I_T \rangle$ is recursively defined as follows:

1. if $\phi = \top$, then $\nu_{\mathcal{I}}(w, \phi) = 1$;
2. if $\phi = \bot$, then $\nu_{\mathcal{I}}(w, \phi) = 0$;
3. if $\phi = v$ is an atom, then $\nu_{\mathcal{I}}(w, \phi) = 1$ if $v \in I_w$, otherwise $\nu_{\mathcal{I}}(w, \phi) = 0$;
4. if $\phi = \neg\psi$, then $\nu_{\mathcal{I}}(w, \phi) = 1$ if, for every world u with $w \leq u$, $\nu_{\mathcal{I}}(u, \psi) = 0$, otherwise $\nu_{\mathcal{I}}(w, \phi) = 0$;
5. if $\phi = (\phi_1 \wedge \phi_2)$, then $\nu_{\mathcal{I}}(w, \phi) = 1$ if $\nu_{\mathcal{I}}(w, \phi_1) = 1$ and $\nu_{\mathcal{I}}(w, \phi_2) = 1$, otherwise $\nu_{\mathcal{I}}(w, \phi) = 0$;
6. if $\phi = (\phi_1 \vee \phi_2)$, then $\nu_{\mathcal{I}}(w, \phi) = 1$ if $\nu_{\mathcal{I}}(w, \phi_1) = 1$ or $\nu_{\mathcal{I}}(w, \phi_2) = 1$, otherwise $\nu_{\mathcal{I}}(w, \phi) = 0$;
7. if $\phi = (\phi_1 \rightarrow \phi_2)$, then $\nu_{\mathcal{I}}(w, \phi) = 1$ if for every world u with $w \leq u$, $\nu_{\mathcal{I}}(u, \phi_1) = 0$ or $\nu_{\mathcal{I}}(u, \phi_2) = 1$, otherwise $\nu_{\mathcal{I}}(w, \phi) = 0$.

We say that ϕ is *true under* $\mathcal{I}$ *in* w iff $\nu_{\mathcal{I}}(w, \phi) = 1$, otherwise ϕ is *false under* $\mathcal{I}$ *in* w. An HT-interpretation $\mathcal{I} = \langle I_H, I_T \rangle$ *satisfies* ϕ, or $\mathcal{I}$ is an *HT-model* of ϕ, iff $\nu_{\mathcal{I}}(H, \phi) = 1$. If ϕ is true under any HT-interpretation, then ϕ is *valid in the logic of here-and-there*, or simply *HT-valid*.

Let S be a set of formulas. An HT-interpretation $\mathcal{I}$ is an HT-model of S iff $\mathcal{I}$ is an HT-model of each element of S. We say that $\mathcal{I}$ is an HT-model of a *program* Π iff $\mathcal{I}$ is an HT-model of $\hat{\Pi} = \{B(r) \rightarrow H(r) \mid r \in \Pi\}$.

Two sets of formulas are *equivalent in the logic of here-and-there*, or *HT-equivalent*, iff they possess the same HT-models. Two formulas, ϕ and ψ, are HT-equivalent iff the sets $\{\phi\}$ and $\{\psi\}$ are HT-equivalent.

It is easily seen that any HT-valid formula is valid in classical logic, but the converse does not always hold. For instance, $p \vee \neg p$ and $\neg\neg p \rightarrow p$ are valid in classical logic but not in the logic of here-and-there as the pair $\langle \emptyset, \{p\} \rangle$ is not an HT-model for either of these formulas.

Equilibrium logic can be seen as a particular type of reasoning with minimal HT-models. Formally, an *equilibrium model* of a formula ϕ is a total HT-interpretation $\langle I, I \rangle$ such that (i) $\langle I, I \rangle$ is an HT-model of ϕ, and (ii) for every proper subset J of I, $\langle J, I \rangle$ is not an HT-model of ϕ.

The following result establishes the close connection between equilibrium models and answer sets, showing that answer sets are actually a special case of equilibrium models:

Proposition 1 ([28, 23]). *For any program Π, I is an answer set of Π iff $\langle I, I \rangle$ is an equilibrium model of $\hat{\Pi}$.*

Moreover, HT-equivalence was shown to capture the notion of strong equivalence between logic programs:

Proposition 2 ([23]). *Let Π_1 and Π_2 be programs, and let $\hat{\Pi}_i = \{B(r) \rightarrow H(r) \mid r \in \Pi_i\}$, for $i = 1, 2$. Then, Π_1 and Π_2 are strongly equivalent iff $\hat{\Pi}_1$ and $\hat{\Pi}_2$ are equivalent in the logic of here-and-there.*

Recently, de Jongh and Hendriks [5] have extended Proposition 2 by showing that for nested programs strong equivalence is characterised precisely by equivalence in all intermediate logics lying between here-and-there (upper bound) and the logic KC of weak excluded middle (lower bound) which is axiomatised by intuitionistic logic together with the schema $\neg\varphi \vee \neg\neg\varphi$. Also, in [32] a (polynomial-time constructible) translation is given which reduces the problem of deciding whether two nested programs are strongly equivalent into the validity problem of classical propositional logic (a similar result was independently shown in [25] for disjunctive programs). As a consequence, checking whether two programs are strongly equivalent has co-NP complexity.

We require the following additional concepts. By an *HT-literal*, l, we understand a formula of form v, $\neg v$, or $\neg\neg v$, where v is a propositional atom or one of $\top$, $\bot$. Furthermore, a formula is in *here-and-there negational normal form*, or *HT-NNF*, if it is made up of HT-literals, conjunctions and disjunctions. Likewise, we say that a *program* is in HT-NNF iff all heads and bodies of rules in the program are in HT-NNF.

Following [24], every expression ϕ can effectively be transformed into an expression ψ in HT-NNF possessing the same HT-models as ϕ. In fact, we have the following property:

Proposition 3. *Every expression ϕ is HT-equivalent to an expression $\nu(\phi)$ in HT-NNF, where $\nu(\phi)$ is constructible in polynomial time from ϕ, satisfying the following conditions, for each expression φ, ψ:*

1. *$\nu(\varphi) = \varphi$, if φ is an HT-literal;*
2. *$\nu(\neg\neg\neg\varphi) = \nu(\neg\varphi)$;*

3. $\nu(\varphi \circ \psi) = \nu(\varphi) \circ \nu(\psi)$, *for* $\circ \in \{\wedge, \vee\}$;
4. $\nu(\neg(\varphi \wedge \psi)) = \nu(\neg\varphi) \vee \nu(\neg\psi)$;
5. $\nu(\neg(\varphi \vee \psi)) = \nu(\neg\varphi) \wedge \nu(\neg\psi)$.

3 Faithful Translations

Next, we introduce the general requirements we impose on our desired translation from nested logic programs into disjunctive logic programs. The following definition is central:

Definition 1. *Let* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *be two alphabets such that* $\mathcal{A}_1 \subseteq \mathcal{A}_2$, *and, for* $i = 1, 2$, *let* $S_i \subseteq NLP_{\mathcal{A}_i}$ *be a class of nested logic programs closed under unions.*[1] *Then, a function* $\rho : S_1 \to S_2$ *is*

1. polynomial *iff, for all programs* $\Pi \in S_1$, *the time required to compute* $\rho(\Pi)$ *is polynomial in the size of* Π;
2. faithful *iff, for all programs* $\Pi \in S_1$,

$$AS_{\mathcal{A}_1}(\Pi) = \{I \cap \mathcal{A}_1 \mid I \in AS_{\mathcal{A}_2}(\rho(\Pi))\};$$

3. strongly faithful *iff, for all programs* $\Pi \in S_1$ *and all programs* $\Pi' \in NLP_{\mathcal{A}_1}$,

$$AS_{\mathcal{A}_1}(\Pi \cup \Pi') = \{I \cap \mathcal{A}_1 \mid I \in AS_{\mathcal{A}_2}(\rho(\Pi) \cup \Pi')\}; \quad \textit{and}$$

4. modular *iff, for all programs* $\Pi_1, \Pi_2 \in S_1$,

$$\rho(\Pi_1 \cup \Pi_2) = \rho(\Pi_1) \cup \rho(\Pi_2).$$

In view of the requirement that $\mathcal{A}_1 \subseteq \mathcal{A}_2$, the general functions considered here may introduce new atoms. Clearly, if the given function is polynomial, the number of newly introduced atoms is also polynomial. Faithfulness guarantees that we can recover the stable models of the input program from the translated program. Strong faithfulness, on the other hand, states that we can add to a given program Π *any nested logic program* Π' and still retain, up to the original language, the semantics of the combined program $\Pi \cup \Pi'$ from $\rho(\Pi) \cup \Pi'$. Finally, modularity enforces that we can translate programs rule by rule.

It is quite obvious that any strongly faithful function is also faithful. Furthermore, strong faithfulness of function ρ implies that, for a given program Π, we can translate *any program part* Π_0 of Π whilst leaving the remaining part $\Pi \setminus \Pi_0$ unchanged, and determine the semantics of Π from $\rho(\Pi_0) \cup (\Pi \setminus \Pi_0)$. As well, for any function of form $\rho : NLP_{\mathcal{A}} \to NLP_{\mathcal{A}}$, strong faithfulness of ρ is equivalent to the condition that Π and $\rho(\Pi)$ are strongly equivalent, for any $\Pi \in NLP_{\mathcal{A}}$. Hence, strong faithfulness generalises strong equivalence.

Following [18, 19], we say that a function ρ as in Definition 1 is *PFM*, or that ρ is a *PFM-function*, iff it is polynomial, faithful, and modular. Analogously,

[1] A class S of sets is closed under unions providing $A, B \in S$ implies $A \cup B \in S$.

we call ρ *PSM*, or a *PSM-function*, iff it is polynomial, strongly faithful, and modular.

It is easy to see that the composition of two PFM-functions is again a PFM-function; and likewise for PSM-functions. Furthermore, since any PSM-function is also PFM, in the following we focus on PSM-functions. In fact, in the next section, we construct a function $\sigma : NLP_{\mathcal{A}_1} \to DLP_{\mathcal{A}_2}$ (where $\mathcal{A}_2$ is a suitable extension of $\mathcal{A}_1$) which is PSM.

Next, we discuss some sufficient conditions guaranteeing that certain classes of functions are strongly faithful. We start with the following concept.

Definition 2. *Let $\rho : NLP_{\mathcal{A}_1} \to NLP_{\mathcal{A}_2}$ be a function such that $\mathcal{A}_1 \subseteq \mathcal{A}_2$, and let $INT_{\mathcal{A}_i}$ be the class of all HT-interpretations over $\mathcal{A}_i$ $(i = 1, 2)$.*

Then, the function $\alpha_\rho : INT_{\mathcal{A}_1} \times NLP_{\mathcal{A}_1} \to INT_{\mathcal{A}_2}$ is called a ρ-associated HT-embedding *iff, for each HT-interpretation $\mathcal{I} = \langle I_H, I_T\rangle$ over $\mathcal{A}_1$, each $\Pi \in NLP_{\mathcal{A}_1}$, and each $w \in \{H, T\}$, $J_w \cap \mathcal{A}_1 = I_w$ and $J_w \setminus \mathcal{A}_1 \subseteq var(\rho(\Pi))$, where $\alpha_\rho(\mathcal{I}, \Pi) = \langle J_H, J_T\rangle$.*

Furthermore, for any $G \subseteq INT_{\mathcal{A}_1}$ and any $\Pi \in NLP_{\mathcal{A}_1}$, we define $\alpha_\rho(G, \Pi) = \{\alpha_\rho(\mathcal{I}, \Pi) \mid \mathcal{I} \in G\}$.

Intuitively, a ρ-associated HT-embedding transforms HT-interpretations over the input alphabet $\mathcal{A}_1$ of ρ into HT-interpretations over the output alphabet $\mathcal{A}_2$ of ρ such that the truth values of the atoms in $\mathcal{A}_1$ are retained. The following definition strengthens these kinds of mappings:

Definition 3. *Let ρ be as in Definition 2, and let α_ρ be a ρ-associated HT-embedding. We say that α_ρ is a* ρ-associated HT-homomorphism *if, for any $\mathcal{I}, \mathcal{I}' \in INT_{\mathcal{A}_1}$ and any $\Pi \in NLP_{\mathcal{A}_1}$, the following conditions hold:*

1. *$\mathcal{I}$ is an HT-model of Π iff $\alpha_\rho(\mathcal{I}, \Pi)$ is an HT-model of $\rho(\Pi)$;*
2. *$\mathcal{I}$ is total iff $\alpha_\rho(\mathcal{I}, \Pi)$ is total;*
3. *if $\mathcal{I} = \langle I_H, I_T\rangle$ and $\mathcal{I}' = \langle I'_H, I'_T\rangle$ are HT-models of Π, then $I_H \subset I'_H$ and $I_T = I'_T$ holds precisely if $J_H \subset J'_H$ and $J_T = J'_T$, for $\alpha_\rho(\mathcal{I}, \Pi) = \langle J_H, J_T\rangle$ and $\alpha_\rho(\mathcal{I}', \Pi) = \langle J'_H, J'_T\rangle$; and*
4. *an HT-interpretation $\mathcal{J}$ over $var(\rho(\Pi))$ is an HT-model of $\rho(\Pi)$ only if $\mathcal{J} \in \alpha_\rho(INT_{\mathcal{A}_1}, \Pi)$.*

Roughly speaking, ρ-associated HT-homomorphisms retain the relevant properties of HT-interpretations for being equilibrium models with respect to transformation ρ. More specifically, the first three conditions take semantical and set-theoretical properties into account, respectively, whilst the last one expresses a specific "closure condition". The inclusion of the latter requirement is explained by observation that the first three conditions alone are not sufficient to exclude the possibility that there may exist some equilibrium model $\mathcal{I}$ of Π such that $\alpha_\rho(\mathcal{I}, \Pi)$ is not an equilibrium model of $\rho(\Pi)$. The reason for this is that the set $\alpha_\rho(INT_{\mathcal{A}_1}, \Pi)$, comprising the images of all HT-interpretations over $\mathcal{A}_1$ under α_ρ with respect to program Π, does, in general, not cover *all* HT-interpretations over $var(\rho(\Pi))$. Hence, for a general ρ-associated HT-embedding $\alpha_\rho(\cdot, \cdot)$, there

may exist some HT-model of $\rho(\Pi)$ which is not included in $\alpha_\rho(INT_{\mathcal{A}_1}, \Pi)$ preventing $\alpha_\rho(\mathcal{I}, \Pi)$ from being an equilibrium model of $\rho(\Pi)$ albeit $\mathcal{I}$ is an equilibrium model of Π. The addition of the last condition in Definition 3, however, excludes this possibility, ensuring that all relevant HT-interpretations required for checking whether $\alpha_\rho(\mathcal{I}, \Pi)$ is an equilibrium model of $\rho(\Pi)$ are indeed considered. The following result can be shown:

Lemma 1. *For any function $\rho : NLP_{\mathcal{A}_1} \to NLP_{\mathcal{A}_2}$ with $\mathcal{A}_1 \subseteq \mathcal{A}_2$, if there is some ρ-associated HT-homomorphism, then ρ is faithful.*

From this, we obtain the following property:

Theorem 1. *Under the circumstances of Lemma 1, if ρ is modular and there is some ρ-associated HT-homomorphism, then ρ is strongly faithful.*

We make use of the last result for showing that the translation from nested logic programs into disjunctive logic programs, as discussed next, is PSM.

4 Main Construction

In this section, we show how logic programs with nested expressions can be efficiently mapped to disjunctive logic programs, preserving the semantics of the respective programs. Although results by Lifschitz *et al.* [24] already provide a reduction of nested logic programs into disjunctive ones (by employing additional transformation steps as given in [19]), that method is exponential in the worst case. This is due to the fact that the transformation relies on distributive laws, yielding an exponential increase of program size whenever the given program contains rules whose heads are in disjunctive normal form or whose bodies are in conjunctive normal form, and the respective expressions are not simple disjunctions or conjunctions of HT-literals.

To avoid such an exponential blow-up, our technique is based on the introduction of new atoms, called *labels*, abbreviating subformula occurrences. This method is derived from structure-preserving normal form translations [36, 33], which are frequently applied in the context of automated reasoning (cf., e.g., [2, 15] for general investigations about structure-preserving normal form translation in finite-valued Gödel logics, and [6, 7] for proof-theoretical issues of such translations for classical and intuitionistic logic). In contrast to theorem proving applications, where the main focus is to provide translations which are satisfiability (or, alternatively, validity) equivalent, here we are interested in somewhat stronger equivalence properties, viz. in the *reconstruction of the answer sets* of the original programs from the translated ones, which involves also an adequate handling of additional minimality criteria.

The overall structure of our translation can be described as follows. Given a nested logic program Π, we perform the following steps:

1. For each $r \in \Pi$, transform $H(r)$ and $B(r)$ into HT-NNF;
2. translate the program into a program containing only rules with conjunctions of HT-literals in their bodies and disjunctions of HT-literals in their heads;
3. eliminate double negations in bodies and heads; and
4. transform the resulting program into a disjunctive logic program, i.e., make all heads negation free.

Steps 1 and 3 are realised by using properties of logic programs as described in [24]; Step 2 represents the central part of our construction; and Step 4 exploits a procedure due to Janhunen [19].

In what follows, for any alphabet $\mathcal{A}$, we define the following new and disjoint alphabets:

- a set $\mathcal{A}_{\mathbf{L}} = \{\mathbf{L}_\phi \mid \phi \in \mathcal{L}_\mathcal{A}\}$ of labels; and
- a set $\bar{\mathcal{A}} = \{\overline{p} \mid p \in \mathcal{A}\}$ of atoms representing negated atoms.

Furthermore, $NLP^{nnf}_{\mathcal{A}}$ is the class of all nested logic programs over $\mathcal{A}$ which are in HT-NNF, and $GDLP^{ht}_{\mathcal{A}}$ is the class of all programs over $\mathcal{A}$ which are defined like generalised logic programs, except that HT-literals may occur in rules instead of ordinary literals.

We assume that for each of the above construction stages, Step i is realized by a corresponding function $\sigma_i(\cdot)$ $(i = 1, \ldots, 4)$. The overall transformation is then described by the composed function $\sigma = \sigma_4 \circ \sigma_3 \circ \sigma_2 \circ \sigma_1$, which is a mapping from the set $NLP_\mathcal{A}$ of all programs over $\mathcal{A}$ into the set $DLP_{\mathcal{A}^*}$ of all disjunctive logic program over $\mathcal{A}^* = \mathcal{A} \cup \mathcal{A}_{\mathbf{L}} \cup \bar{\mathcal{A}}$. More specifically,

$$\sigma_1 : NLP_\mathcal{A} \to NLP^{nnf}_{\mathcal{A}}$$

translates any nested logic program over $\mathcal{A}$ into a nested program in HT-NNF. Translation

$$\sigma_2 : NLP^{nnf}_{\mathcal{A}} \to GDLP^{ht}_{\mathcal{A}\cup\mathcal{A}_{\mathbf{L}}}$$

takes these programs and transforms their rules into simpler ones as described by Step 2, introducing new labels. These rules are then fed into mapping

$$\sigma_3 : GDLP^{ht}_{\mathcal{A}\cup\mathcal{A}_{\mathbf{L}}} \to GDLP_{\mathcal{A}\cup\mathcal{A}_{\mathbf{L}}},$$

yielding generalised disjunctive logic programs. Finally,

$$\sigma_4 : GDLP_{\mathcal{A}\cup\mathcal{A}_{\mathbf{L}}} \to DLP_{\mathcal{A}^*}$$

outputs standard disjunctive logic programs.

As argued in the following, each of these functions is PSM; hence, the overall function $\sigma = \sigma_4 \circ \sigma_3 \circ \sigma_2 \circ \sigma_1$ is PSM as well.

We continue with the technical details, starting with σ_1.

For the first step, we use the procedure $\nu(\cdot)$ from Proposition 3 to transform heads and bodies of rules into HT-NNF.

Definition 4. *The function* $\sigma_1 : NLP_{\mathcal{A}} \to NLP_{\mathcal{A}}^{nnf}$ *is defined by setting*

$$\sigma_1(\Pi) = \{\nu(H(r)) \leftarrow \nu(B(r)) \mid r \in \Pi\},$$

for any $\Pi \in NLP_{\mathcal{A}}$.

Since, for each expression ϕ, $\nu(\phi)$ is constructible in polynomial time and ϕ is HT-equivalent to $\nu(\phi)$ (cf. Proposition 3), the following result is immediate:

Lemma 2. *The translation* σ_1 *is PSM.*

The second step is realised as follows:

Definition 5. *The function* $\sigma_2 : NLP_{\mathcal{A}}^{nnf} \to GDLP_{\mathcal{A}\cup\mathcal{A}_{\mathbf{L}}}^{ht}$ *is defined by setting, for any* $\Pi \in NLP_{\mathcal{A}}^{nnf}$,

$$\sigma_2(\Pi) = \{\mathbf{L}_{H(r)} \leftarrow \mathbf{L}_{B(r)} \mid r \in \Pi\} \cup \gamma(\Pi),$$

where $\gamma(\Pi)$ *is constructed as follows:*

1. *for each HT-literal* l *occurring in* Π*, add the two rules*

$$\mathbf{L}_l \leftarrow l \quad \text{and} \quad l \leftarrow \mathbf{L}_l;$$

2. *for each expression* $\phi = (\phi_1 \wedge \phi_2)$ *occurring in* Π*, add the three rules*

$$\mathbf{L}_\phi \leftarrow \mathbf{L}_{\phi_1} \wedge \mathbf{L}_{\phi_2}, \quad \mathbf{L}_{\phi_1} \leftarrow \mathbf{L}_\phi, \quad \mathbf{L}_{\phi_2} \leftarrow \mathbf{L}_\phi; \quad \text{and}$$

3. *for each expression* $\phi = (\phi_1 \vee \phi_2)$ *occurring in* Π*, add the three rules*

$$\mathbf{L}_{\phi_1} \vee \mathbf{L}_{\phi_2} \leftarrow \mathbf{L}_\phi, \quad \mathbf{L}_\phi \leftarrow \mathbf{L}_{\phi_1}, \quad \mathbf{L}_\phi \leftarrow \mathbf{L}_{\phi_2}.$$

This definition is basically an adaption of a structure-preserving normal form translation for intuitionistic logic, as described in [26].

It is quite obvious that σ_2 is modular and, for each $\Pi \in NLP_{\mathcal{A}}^{nnf}$, we have that $\sigma_2(\Pi)$ is constructible in polynomial time. In order to show that σ_2 is strongly faithful, we define a suitable HT-homomorphism as follows.

Sublemma 1 *Let* σ_2 *be the translation defined above, and let* $\sigma_2^* : NLP_{\mathcal{A}} \to NLP_{\mathcal{A}\cup\mathcal{A}_{\mathbf{L}}}$ *result from* σ_2 *by setting* $\sigma_2^*(\Pi) = \sigma_2(\Pi)$ *if* $\Pi \in NLP_{\mathcal{A}}^{nnf}$ *and* $\sigma_2^*(\Pi) = \Pi$ *if* $\Pi \in NLP_{\mathcal{A}} \setminus NLP_{\mathcal{A}}^{nnf}$.

Then, the function $\alpha_{\sigma_2^*} : INT_{\mathcal{A}} \times NLP_{\mathcal{A}} \to INT_{\mathcal{A}\cup\mathcal{A}_{\mathbf{L}}}$, *defined as*

$$\alpha_{\sigma_2^*}(\mathcal{I}, \Pi) = \langle I_H \cup \lambda_H(\mathcal{I}, \Pi), I_T \cup \lambda_T(\mathcal{I}, \Pi)\rangle,$$

is a σ_2^**-associated HT-homomorphism, where*

$$\lambda_w(\mathcal{I}, \Pi) = \{\mathbf{L}_\phi \in \mathcal{A}_{\mathbf{L}} \cap var(\sigma_2^*(\Pi)) \mid \nu_{\mathcal{I}}(w, \phi) = 1\}$$

if $\Pi \in NLP_{\mathcal{A}}^{nnf}$, *and* $\lambda_w(\mathcal{I}, \Pi) = \emptyset$ *otherwise, for any* $w \in \{H, T\}$ *and any HT-interpretation* $\mathcal{I} = \langle I_H, I_T\rangle$ *over* $\mathcal{A}$.

Hence, according to Theorem 1, σ_2^* is strongly faithful. As a consequence, σ_2 is strongly faithful as well. Thus, the following holds:

Lemma 3. *The function σ_2 is PSM.*

For Step 3, we use a method due to Lifschitz *et al.* [24] for eliminating double negations in heads and bodies of rules. The corresponding function σ_3 is defined as follows:

Definition 6. *Let $\sigma_3 : GDLP^{ht}_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}} \to GDLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$ be the function obtained by replacing, for each given program $\Pi \in GDLP^{ht}_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$, each rule $r \in \Pi$ of form*

$$\phi \vee \neg\neg p \leftarrow \psi \quad \text{by} \quad \phi \leftarrow \psi \wedge \neg p,$$

as well as each rule of form

$$\phi \leftarrow \psi \wedge \neg\neg q \quad \text{by} \quad \phi \vee \neg q \leftarrow \psi,$$

where ϕ and ψ are expressions and $p, q \in \mathcal{A}$.

As shown in [24], performing replacements of the above type results in programs which are strongly equivalent to the original programs. In fact, it is easy to see that such replacements yield transformed programs which are strongly faithful to the original ones. Since these transformations are clearly modular and constructible in polynomial time, we obtain that σ_3 is PSM.

Lemma 4. *The function σ_3 is PSM.*

Finally, we eliminate remaining negations possibly occurring in the heads of rules. To this end, we employ a procedure due to Janhunen [19] (for an alternative method, cf. [17]).

Definition 7. *Let $\sigma_4 : GDLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}} \to DLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}\cup\bar{\mathcal{A}}}$ be the function defined by setting, for any program $\Pi \in GDLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$,*

$$\sigma_4(\Pi) = \bar{\Pi} \cup \{\bot \leftarrow (p \wedge \overline{p}),\ \overline{p} \leftarrow \neg p \mid \neg p \text{ occurs in the head of some rule in } \Pi\},$$

where $\bar{\Pi}$ results from Π by replacing each occurrence of a literal $\neg p$ in the head of a rule in Π by $\overline{p}$.

Janhunen showed that replacements of the above kind lead to a transformation which is PFM. As a matter of fact, since his notion of faithfulness is somewhat stricter than ours, the results in [19] actually imply that

$$AS_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}(\Pi \cup \Pi') = \{I \cap (\mathcal{A} \cup \mathcal{A}_\mathbf{L}) \mid I \in AS_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}\cup\bar{\mathcal{A}}}(\sigma_4(\Pi) \cup \Pi')\},$$

for any $\Pi, \Pi' \in GDLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$. However, we need a stronger condition here, viz. that the above equation holds for any $\Pi \in GDLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$ and any $\Pi' \in NLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$. We show this by appeal to Theorem 1.

Sublemma 2 *Let σ_4 be the translation defined above, and let $\sigma_4^* : NLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}} \to NLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}\cup\bar{\mathcal{A}}}$ result from σ_4 by setting $\sigma_4^*(\Pi) = \sigma_4(\Pi)$ if $\Pi \in GDLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$ and $\sigma_4^*(\Pi) = \Pi$ if $\Pi \in NLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}} \setminus GDLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$.*

Then, the function $\alpha_{\sigma_4^*} : INT_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}} \times NLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}} \to INT_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}\cup\bar{\mathcal{A}}}$, defined as

$$\alpha_{\sigma_4^*}(\mathcal{I}, \Pi) = \langle I_H \cup \kappa(\mathcal{I}, \Pi), I_T \cup \kappa(\mathcal{I}, \Pi)\rangle,$$

is a σ_4^*-associated HT-homomorphism, where

$$\kappa(\mathcal{I}, \Pi) = \{\overline{p} \mid \neg p \text{ occurs in the head of some rule in } \Pi \text{ and } p \notin I_T\}$$

if $\Pi \in GDLP_{\mathcal{A}\cup\mathcal{A}_\mathbf{L}}$, and $\kappa(\mathcal{I}, \Pi) = \emptyset$ otherwise, for any HT-interpretation $\mathcal{I} = \langle I_H, I_T\rangle$ over $\mathcal{A} \cup \mathcal{A}_\mathbf{L}$.

Observe that, in contrast to the definition of function $\alpha_{\sigma_2^*}$ from Sublemma 1, here the same set of newly introduced atoms is added to both worlds. As before, we obtain that σ_4^* is strongly faithful, and hence that σ_4 is strongly faithful as well.

Lemma 5. *The function σ_4 is PSM.*

Summarising, we obtain our main result, which is as follows:

Theorem 2. *Let $\sigma_1, \ldots, \sigma_4$ be the functions defined above. Then, the composed function $\sigma = \sigma_4 \circ \sigma_3 \circ \sigma_2 \circ \sigma_1$, mapping nested logic programs over alphabet $\mathcal{A}$ into disjunctive logic programs over alphabet $\mathcal{A} \cup \mathcal{A}_\mathbf{L} \cup \bar{\mathcal{A}}$, is polynomial, strongly faithful, and modular.*

Since strong faithfulness implies faithfulness, we get the following corollary:

Corollary 1. *For any nested logic program Π over $\mathcal{A}$, the answer sets of Π are in a one-to-one correspondence to the answer sets of $\sigma(\Pi)$, determined by the following equation:*

$$AS_\mathcal{A}(\Pi) = \{I \cap \mathcal{A} \mid I \in AS_{\mathcal{A}^*}(\sigma(\Pi))\},$$

where $\mathcal{A}^* = \mathcal{A} \cup \mathcal{A}_\mathbf{L} \cup \bar{\mathcal{A}}$.

We conclude with a remark concerning the construction of function σ_2. As pointed out previously, this mapping is based on a structure-preserving normal form translation for intuitionistic logic, as described in [26]. Besides the particular type of translation used here, there are also other, slightly improved structure-preserving normal form translations in which fewer rules are introduced, depending on the polarity of the corresponding subformula occurrences. However, although such optimised methods work in monotonic logics, they are not sufficient in the present setting. For instance, in a possible variant of translation σ_2 based on the polarity of subformula occurrences, instead of introducing all three rules for an expression ϕ of form $(\phi_1 \wedge \phi_2)$, only $\mathbf{L}_\phi \leftarrow \mathbf{L}_{\phi_1} \wedge \mathbf{L}_{\phi_2}$ is used if ϕ occurs in the body of some rule, or both $\mathbf{L}_{\phi_1} \leftarrow \mathbf{L}_\phi$ and $\mathbf{L}_{\phi_2} \leftarrow \mathbf{L}_\phi$ are used if ϕ occurs in the head of some rule, and analogous manipulations are performed for atoms and disjunctions. Applying such an encoding to $\Pi = \{p \leftarrow;\ q \leftarrow;\ r \vee (p \wedge q) \leftarrow\ \}$ over $\mathcal{A}_0 = \{p, q, r\}$ yields a translated program possessing two answer sets, say S_1 and S_2, such that $S_1 \cap \mathcal{A}_0 = \{p, q\}$ and $S_2 \cap \mathcal{A}_0 = \{p, q, r\}$, although only $\{p, q\}$ is an answer set of Π.

5 Conclusion

We have developed a translation of logic programs with nested expressions into disjunctive logic programs. We have proven that our translation is polynomial, strongly faithful, and modular. This allows us to utilise off-the-shelf disjunctive logic programming systems for interpreting nested logic programs. In fact, we have implemented our translation as a front end for the system `DLV` [8, 9]. The corresponding compiler is implemented in Prolog and can be downloaded from the Web at URL

`http://www.cs.uni-potsdam.de/~torsten/nlp`.

Our technique is based on the introduction of new atoms, abbreviating subformula occurrences. This method has its roots in structure-preserving normal form translations [36, 33], which are frequently used in automated deduction. In contrast to theorem proving applications, however, where the main focus is to provide satisfiability (or, alternatively, validity) preserving translations, we are concerned with much stronger equivalence properties, involving additional minimality criteria, since our goal is to *reconstruct* the answer sets of the original programs from the translated ones.

With the particular labeling technique employed here, our translation avoids the risk of an exponential blow-up in the worst-case, as faced by a previous approach of Lifschitz *et al.* [24] due to the usage of distributivity laws. However, this is not to say that our translation is *always* the better choice. As in classical theorem proving, it is rather a matter of experimental studies under which circumstances which approach is the more appropriate one. To this end, besides the implementation of our structural translation, we have also implemented the distributive translation into disjunctive logic programs in order to conduct experimental results. These experiments are subject to current research.

Also, we have introduced the concept of *strong faithfulness*, as a generalisation of (standard) faithfulness and strong equivalence. This allows us, for instance, to translate, in a semantics-preserving way, arbitrary program parts and leave the remaining program unaffected.

Acknowledgements

This work was partially supported by the German Science Foundation (DFG) under grant FOR 375/1-1, TP C, as well as by the Austrian Science Fund (FWF) under grants P15068-INF and N Z29-INF. The authors would like to thank Agata Ciabattoni for pointing out some relevant references.

References

[1] M. Baaz, U. Egly, and A. Leitsch. Normal Form Transformations. In *Handbook of Automated Reasoning*, volume I, chapter 5, pages 273–333. Elsevier Science B. V., 2001. 406

[2] M. Baaz and C. G. Fermüller. Resolution-based Theorem Proving for Many-valued Logics. *Journal of Symbolic Computation*, 19(4):353–391, 1995. 407, 413

[3] C. Baral and C. Uyan. Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In *Proc. LPNMR-01*, pages 186–199, 2001. 406

[4] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proc. AAAI-98*, pages 262–267, 1998. 407

[5] D. de Jongh and L. Hendriks. Characterization of Strongly Equivalent Logic Programs in Intermediate Logics. Technical report, 2001. Preprint at http://turing.wins.uva.nl/~lhendrik/. 410

[6] U. Egly. On Different Structure-Preserving Translations to Normal Form. *Journal of Symbolic Computation*, 22(2):121–142, 1996. 413

[7] U. Egly. On Definitional Transformations to Normal Form for Intuitionistic Logic. *Fundamenta Informaticae*, 29(1,2):165–201, 1997. 413

[8] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Non-monotonic Reasoning. In *Proc. LPNMR-97*, pages 363–374, 1997. 406, 418

[9] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System dlv: Progress Report, Comparisons and Benchmarks. In *Proc. KR-98*, pages 406–417, 1998. 406, 418

[10] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulas. In *Proc. AAAI-00*, pages 285–290, 2000. 407

[11] M. Gelfond, M. Balduccini, and J. Galloway. Diagnosing Physical Systems in A-Prolog. In *Proc. LPNMR-01*, pages 213–225, 2001. 406

[12] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991. 405

[13] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In *Proc. IJCAR-01*, pages 364–369, 2001. 407

[14] K. Gödel. Zum intuitionistischen Aussagenkalkül. *Anzeiger der Akademie der Wissenschaften in Wien*, pages 65–66, 1932. 407

[15] R. Hähnle. Short Conjunctive Normal Forms in Finitely Valued Logics. *Journal of Logic and Computation*, 4(6):905–927, 1994. 407, 413

[16] K. Heljanko and I. Niemelä. Bounded LTL Model Checking with Stable Models. In *Proc. LPNMR-01*, pages 200–212, 2001. 406

[17] K. Inoue and C. Sakama. Negation as Failure in the Head. *Journal of Logic Programming*, 35(1):39–78, 1998. 416

[18] T. Janhunen. On the Intertranslatability of Autoepistemic, Default and Priority Logics, and Parallel Circumscription. In *Proc. JELIA-98*, pages 216–232, 1998. 411

[19] T. Janhunen. On the Effect of Default Negation on the Expressiveness of Disjunctive Rules. In *Proc. LPNMR-01*, pages 93–106, 2001. 407, 411, 413, 414, 416

[20] H. Kleine-Büning, M. Karpinski, and A. Flögel. Resolution for Quantified Boolean Formulas. *Information and Computation*, 117(1):12–18, 1995. 407

[21] R. Letz. Advances in Decision Procedures for Quantified Boolean Formulas. In *Proc. IJCAR-01 Workshop on Theory and Applications of Quantified Boolean Formulas*, pages 55–64, 2001. 407

[22] V. Lifschitz. Answer Set Planning. In *Proc. ICLP-99*, pages 23–37, 1999. 406

[23] V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001. 406, 407, 408, 409, 410

[24] V. Lifschitz, L. Tang, and H. Turner. Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999. 405, 407, 408, 410, 413, 414, 416, 418

[25] F. Lin. Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic. In *Proc. KR-02*, pages 170–176, 2002. 410

[26] G. Mints. Resolution Strategies for the Intuitionistic Logic. In *Constraint Programming: NATO ASI Series*, pages 282–304. Springer, 1994. 406, 415, 417

[27] I. Niemelä and P. Simons. Smodels: An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In *Proc. LPNMR-97*, pages 420–429, 1997. 406

[28] D. Pearce. A New Logical Characterisation of Stable Models and Answer Sets. In *Non-Monotonic Extensions of Logic Programming*, pages 57–70. Springer, 1997. 406, 407, 409, 410

[29] D. Pearce. From Here to There: Stable Negation in Logic Programming. In *What is Negation?* Kluwer, 1999. 407, 409

[30] D. Pearce, I. de Guzmán, and A. Valverde. A Tableau Calculus for Equilibrium Entailment. In *Proc. TABLEAUX-00*, pages 352–367, 2000. 409

[31] D. Pearce, I. de Guzmán, and A. Valverde. Computing Equilibrium Models Using Signed Formulas. In *Proc. CL-00*, pages 688–702, 2000. 409

[32] D. Pearce, H. Tompits, and S. Woltran. Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In *Proc. EPIA-01*, pages 306–320. Springer, 2001. 406, 407, 410

[33] D. A. Plaisted and S. Greenbaum. A Structure Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. 406, 413, 418

[34] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proc. IJCAI-99*, pages 1192–1197, 1999. 407

[35] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer-Verlag, 1983. 420

[36] G. Tseitin. On the Complexity of Proofs in Propositional Logics. *Seminars in Mathematics*, 8, 1970. Reprinted in [35]. 406, 413, 418

Using Logic Programming to Detect Activities in Pervasive Healthcare

Henrik Bærbak Christensen

Center for Pervasive Computing, University of Aarhus
DK-8200 Århus N, Denmark
Tel.: +45 89 42 32 00
hbc@daimi.au.dk

Abstract. In this experience paper we present a case study in using logic programming in a pervasive computing project in the healthcare domain. An expert system is used to detect healthcare activities in a pervasive hospital environment where positions of people and things are tracked. Based on detected activities an activity-driven computing infrastructure provides computational assistance to healthcare staff on mobile- and pervasive computing equipment. Assistance range from simple activities like fast log-in into the electronic patient medical record system to complex activities like signing for medicine given to specific patients. We describe the role of logic programming in the infrastructure and discuss the benefits and problems of using logic programming in a pervasive context.

1 Introduction

Pervasive computing is a new, interesting, topic in computer science. The promise is to bring computing assistance anywhere and anytime [26]. While the perspectives are fantastic so are the challenges.

In this paper we report from the *pervasive healthcare project* [24] that we are presently working on in the Center for Pervasive Computing, University of Aarhus (CfPC) [8]. The research objectives of the pervasive healthcare project are to experiment with enhancing the quality of everyday healthcare activities utilizing pervasive computing technology. The project is collaboration between CfPC, the Aarhus County Hospital (AAS), and a Danish company that is developing an electronic patient medical record (EPR) system for Aarhus county.

At present, patient medical records are paper-based at the county hospital. Paper records have some inherent problems. Bad handwriting introduces errors, repetitive data must be manually copied, records are difficult to keep up-to-date, records get lost, and also a significant amount of time is spent simply finding them as the records are carried around a lot.

An electronic patient medical record system overcomes many of the data loss and consistency problems. However, new problems arise. Mobility and easy access are primary advantages of paper records. In contrast, laptop computers are too heavy to carry around; Personal Digital Assistants (PDAs) have very small

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 421–436, 2002.
© Springer-Verlag Berlin Heidelberg 2002

screens; and stationary computers must enforce log-in and log-out procedures to ensure data security and privacy and thus substantial time is wasted constantly keying in our username and password and finding patient data.

In the healthcare project, we have designed an *activity-driven computing infrastructure* where everyday healthcare activities define the basic computational services provided for the staff. It has been designed in collaboration with nurses and doctors from Aarhus county hospital and evaluated at workshops. Central to this infrastructure is the *activity discovery component* that is an expert system that monitors movement of people and things in the environment and combines this information with context information and heuristics about work processes to guess at occurring activities.

While the project's main research objective is the study of architectures for pervasive computing, rule-based and logic programming turned out to be a strong and natural paradigm for detecting human activity in a pervasive computing environment. This insight and our experiences are the contribution of the present paper while architectural- and user interface aspects will be reported elsewhere [4,3,2,10].

2 Setting the Stage

Within CfPC we conduct research in an experimental and multidisciplinary manner with participation of industrial partners. Our project team consists of computer scientists with various backgrounds: Computer supported collaborative work (CSCW), human-computer interaction (HCI), software architecture, and distributed computing, as well as industrial developers, an ethnographer, and clinicians from the hospital.

Our research focus is primarily directed in two directions, namely software architectures to support healthcare in a pervasive and ubiquitous computing environment and CSCW and HCI issues in this context. Thus, we had no plans to venture into the area of logic programming at the start up of our project.

Our research methods include ethnographic observations of clinical work [2] and scenario-based design methods [7,1]. A cornerstone in our design validation effort is *workshops* in which clinicians perform role-playing games of future work situations using our prototypes to test their feasibility in semi-realistic situations. These prototypes are characterized by a number of properties:

- *Limited functionality:* Functionality is usually limited; typically we only implement just what is required to role-play a fixed number of scenarios. For example, our prototype only deals with activities concerning the medicine schema of three patients; no other medical record data is included and other patient care activities are disregarded.
- *Limited datasets:* The data to be used in the role-plays is usually hard-coded into the prototype or read from simple files instead of utilizing e.g. database technology. The size of data used is limited, for instance our prototype knows about three nurses, two doctors, three patients, and two medicine trays.

Again, this limited dataset suffices for role-playing a number of work situations for a known set of users.

The basic premise of prototypes is to validate whether the underlying functionality- and usage principles are sound in the given context, before addressing architectural qualities such as performance, modifiability, scalability, etc. The point is that a high-performance, reliable, and secure system is not interesting if it is impossible to use or if it does not solve the right problems for the users.

These premises are important to understand in our discussion. Our main contribution is that we find logic programming approaches promising in computing contexts that are centered around human activities (in contrast to the prevailing document-centered paradigm known from the office environment); but still a lot of issues remains to be investigated further in a realistic deployment situation with respect to scalability and performance.

3 An Activity-Driven Healthcare Scenario

To given an idea of the functionality of our activity-driven computing infrastructure a small example may be helpful.

3.1 Pervasive Environment

We assume some kind of pervasive infrastructure is already in place in the healthcare environment. Specifically, we assume:

- *Computing devices are readily accessible.* We envision that clinicians may be carrying PDAs and/or tablet computers. Laptop quality computers are built into hospital beds (perhaps with touch sensitive screens and without keyboards). Very large computer screens are built into walls of conference rooms. All devices are connected in a reliable and high bandwidth network.
- *Location-Awareness.* All persons and relevant artifacts wear devices that allow the computing infrastructure to monitor their movements and their location.

Given such a pervasive healthcare environment the following scenario illustrates the type of activities our computing infrastructure is able to infer.

3.2 Scenario

Nurse Mrs. Andersen is going to give the 12 o'clock medicine to her patients. She carries the medicine trays of patients Mrs. Hansen and Mr. Jensen. She approaches Mrs. Hansen that lies in her bed and puts down Hansen's medicine tray on the table next to the bed's touch sensitive computer screen.

As the activity-driven computing infrastructure constantly monitors movements of people and things, it detects that a) nurse Andersen is near Hansen's bed, b) Mrs. Hansen is also near the bed, and c) Hansen's medicine tray is near

the bed. From fact a) it infers a likely activity to simply "log nurse Andersen into EPR on Hansen's bed computer". From fact a) and b) it guesses at an activity "log into EPR and fetch the patient record for Mrs. Hansen". Furthermore it combines facts a), b), and c) with context data and heuristics: as the time is around noon and the medical record shows that today's 12 o'clock medicine has not yet been given to Hansen, it infers two additional activities "log into EPR and show today's medicine schema for Hansen" and "log into EPR and record all 12 o'clock medicine for today taken by Hansen, signed by nurse Andersen". The former activity fetches all relevant data for the nurse but does not change any data in EPR; in contrast the latter activity is a "shortcut" that both fetches the data and also records in EPR that the prescribed medicine has indeed been taken by the patient.

The four activities are forwarded to Hansen's bed computer and appear as four buttons on a dedicated *activity bar*. The activity bar is akin the task bar known from the Windows platform. The activity bar is always visible but does not visually nor operationally disrupt the computer display. Nurse Andersen clicks the button marked "show today's medicine schema for Hansen". This *activates* the activity, which means it is forwarded to the EPR system where it is enacted: the medicine schema data is fetched and the user interface formatted to show the proper part of the schema. However, before she can sign for any medicine, she is interrupted by a phone call and leaves the room. As the nurse is no longer in the vicinity of the bed, the activities are removed from the activity bar and the EPR system is closed on Hansen's bed computer to avoid confidential data to be seen.

Two minutes later, another nurse, Mr. Christensen, arrives at Hansen's bed. The infrastructure now performs the same computation as outlined before, and thus forwards four activities to the activity bar with the important change that log-in and signing will be on behalf of nurse Christensen, of course. The nurse asks Mrs. Hansen if she has taken her pills and, as Mrs. Hansen confirms, he chooses the "record all 12 o'clock medicine for today taken by Hansen, signed by nurse Christensen" activity that enacts the required changes in EPR.

The scenario hopefully shows two important benefits. First, proposing activities saves times in the daily healthcare work: a lot of tedious typing and user interface navigation is avoided. Secondly, the attention is moved from handling computers to the real issue: the patients.

4 Activity-Driven Computing Infrastructure

A logical view architecture diagram is shown in Fig. 1. The major components and their responsibilities are:

– *Tag Scanner, WLAN Monitor, BlueTooth:* These components handles the hardware and generate location/movement events that are sent to the location server.

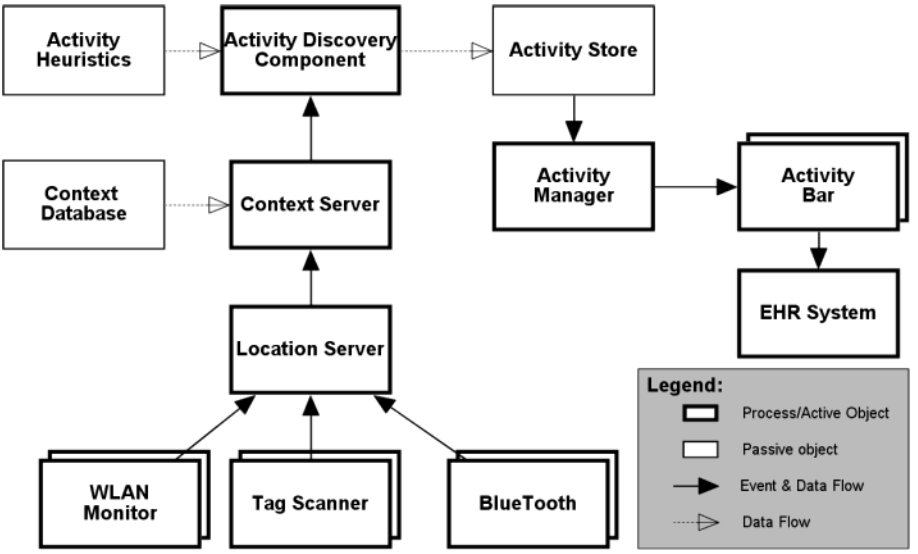


Fig. 1. Logical view of infrastructure

- *Location Server:* Receives events, like tag-enter and tag-leave, from the hardware and maps hardware IDs to logical IDs. Events are sent to the context server.
- *Context Server:* Maps logical IDs to physical location information based on knowledge of the physical location of scanners and knowledge of what tag any given person or thing is wearing.
- *Activity Discovery Component (ADC):* Infer possible activities based on information from location- and context server and heuristics about recurring activities in healthcare. Once they are created, they are stored in the activity store.
- *Activity Store:* Detected activities, as well as activities explicitly created by healthcare staff, are stored here. Upon storing, the activity manager is notified about new activities.
- *Activity Manager:* Receives notifications of new activities and forwards activities to all pervasive computing equipment (more accurately, the activity bar running on the device) that is near the person that the activity relates to.
- *Activity Bar:* Receives activities and presents them non-intrusively to the healthcare staff. It is a separate application that resembles the task-bar known from the Windows operating system. Activities are not activated before the user explicitly selects them, typically by clicking the icon of the activity on the activity bar. Upon activation the activity is forwarded to the proper application, typically the EPR system, where it fetches relevant data and formats the user interface properly.

– *Electronic Patient Record System:* Third party database and application handling patient record information. Accepts activities from the activity bar and fetches data and formats the user interface according to the specifications of the activity.

Detailed description of our design is provided elsewhere [9].

4.1 Prototype Implementation

Our workshop experiments were conducted using *Radio Frequency IDentity* (RFID) tags. These tags are cheap, weigh a few grams, are paper-thin, and are easily glued onto a medicine tray or worn on a clinician's coat. Each RFID-tag has its own unique 64-bit identity. A *tag scanner* is able to detect the 64-bit identity of a tag whenever it enters the scanners detection area (about 0.5 meters) and also whenever it leaves the detection area again. These events we denote *tag-enter* and *tag-leave* events respectively.

The "pervasive" computing equipment was simulated during workshop experiments by laptop computers. A snapshot from our evaluation workshop is seen in Fig. 2. To the left you see the ICode tag scanner and two tags on top of it. The upper one is taped onto cardboard and is the nurse's personal tag. Below is a medicine tray with a tag glued onto the bottom of it. On the right, partially covered by the nurse's back, is a laptop computer that displays activity-bar and the EPR system and responds to activities guessed by the activity-driven computing infrastructure.

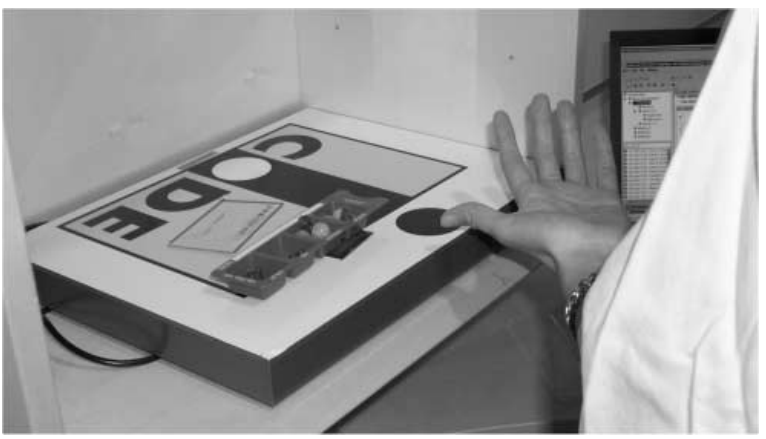


Fig. 2. Snapshot from the workshop showing our prototype RFID based setup

5 Logic Programming for Activity Discovery

Our prototype implementation is made in Java Standard Edition 1.3. The developing team consisted of three experienced object-oriented programmers but with limited knowledge of logic programming.

Detecting activities is cumbersome from a procedural/object-oriented programming paradigm point of view. Activities happen when a number of persons and things meet in time and space and other conditions are met such as the time of day, personal preferences, and the state of patient record data. Many activities are interrelated and interact in complex ways. Our idea was that a rule based inference engine [18] would serve us better for inferring possible activities than writing them in the object-oriented paradigm of Java. Therefore we wanted to experiment with a logic programming (LP) approach in a opportunistic way: our goal was to clarify whether LP would ease and enhance our ability to express and detect human activities in our context; our concern was not really to find the "best" LP system.

A search on the internet lead us to Jess: the Java Expert System Shell [15] which seemed ideal primarily due to its strong and seamless integration with Java. Jess was originally developed as a Java based implementation of CLIPS [11] but has added special features over the years. Before discussing our design in more detail, we will outline Jess and the way we use it.

5.1 Modeling in Jess

Jess is an expert system of the *production system* variant [5]. A production system is defined in terms of *rules* or *productions* together with a database of current *assertions* or *facts*, stored in *working memory* or a *knowledge base*. Note that facts in Jess are *ground* facts, i.e. they do not contain variables.

Rules have two parts, the *left-hand-side* (LHS) and the *right-hand-side* (RHS). The LHS is a conjunction of pattern elements that are matched against the facts in the knowledge base. The RHS contains directives that modify the knowledge base by adding or removing facts and/or have external side effects like invoking Java methods.

This makes production systems very different from Prolog—as stated in the Jess manual: "Prolog is really about answering queries, while Jess is about acting in response to inputs." The latter is exactly what we need in our pervasive healthcare context.

Jess contains data structuring mechanisms that feel familiar to object-oriented programmers: using the *template* construct you can define structured objects with fields (denoted "unordered facts" in Jess) and define single-inheritance hierarchies.

To demonstrate both Jess and the general way we use it, we describe a couple of simple examples. Below is shown how a `Patient` object can be defined:

```
(deftemplate Person
  "A Person class"
  (slot id)
  (slot name)
)
(deftemplate Patient extends Person
  "A Patient class"
  (slot bedId)
)
```

Here a Person template define that facts about Persons each contains a unique identity `id` and a string value `name`. A Patient fact in addition contains the identity of the bed she/he is using. Given these templates you can define a patient in the knowledge base using the `assert` imperative:

```
(assert
  (Patient
   (name "Mr. Hansen")
   (id 1103448675)
   (bedId 5638821)
  )
)
```

A key point in our design is that events from our location- and context-server are modeled by event templates and corresponding facts inserted into the knowledge base whenever they occur.

```
(deftemplate Move
  "Some entity with given id has moved to the given location"
  (slot location)
  (slot id)
)
(deftemplate PersonMove extends Move
  "A person has moved to a given location"
)
(deftemplate EquipmentMove extends Move
  "An equipment of some sort has moved to a given location"
)
```

Note that the inheritance is used simply to classify some move events as either moves of persons or equipment instead of defining type in a slot.

Finally, as a simple example of a rule we can combine facts about a patient and movement of a person:

```
(defrule report-patient-location
  (Patient (id ?id) (name ?name))
  (PersonMove (id ?id) (location ?location))
  =>
  (printout t "Patient " ?name " has moved to location: "
        ?location crlf)
)
```

As the same identifier, `?id`, is used in both pattern elements in the LHS, they have to have identical values in order for the rule to fire. For those familiar with the CHR language [16], the above rule would look quite familiar. This is no surprise since Jess and CHRs are both rule based forward chaining systems, one of the main differences being that while Jess is based on the state preserving RETE match algorithm [14], CHR is based on the state-less TREAT algorithm [21].

Rules are only fired when the inference engine is explicitly activated using the `run` imperative in Jess. Thus, a simple Jess session using the example above may look like this:

```
Jess> (facts)
f-0   (initial-fact)
f-1   (Patient (id 1103448675) (name "Mr. Hansen")
         (bedId 5638821))
For a total of 2 facts.
Jess> (assert (PersonMove (id 1103448675) (location 7)))
<Fact-2>
Jess> (run)
Patient Mr. Hansen has moved to location: 7
```

In our prototype, the RHS typically contains Java method invocations as described in the next section.

5.2 Modeling Healthcare Activities

Our basic idea was to let the ADC have a knowledge base containing facts about persons and equipment, and rules that describe possible activities. Our context server generates `PersonMove` and `EquipmentMove` events and sends these to the ADC. The ADC then inserts these as facts into the knowledge base and runs the inference engine. If any rule fires it will callback into the Java code to generate activity objects for further handling by the activity-driven infrastructure.

For example, in the scenario given in section 3.2 we can describe the rule that infers activity "log into EPR and show today's medicine schema for patient `pttId`" in Jess syntax as:

```
(defrule show-medicine-schema-activity
  (PersonMove (location ?loc) (id ?staffId) )
  (Staff (id ?staffId) )
  (PersonMove (location ?loc) (id ?pttId) )
  (Patient (id ?pttId) )
  (ActivityBarProgram (id ?progid) (location ?loc))
  =>
  (sendShowMedicineSchemaActivity ( ?progid ?staffId ?pttId ) )
)
```

The first two pattern elements ensure that a nurse or doctor is present in location `?loc`; the next two that a patient is present in the same place; and the final element ensures that a computing device with a running activity bar is present. Then the RHS makes a callback to Java that a `ShowMedicineSchemaActivity` must be created with the given parameters.

One of the most complicated rules in our present prototype is shown below:

```
(defrule document-medicine-given
  "handle case where medicine tray is seen at a location and
   the associated patient and a clinician is present"
  (EquipmentMove (location ?loc) (id ?eid) )
  (Tray (id ?eid) (patientId ?pttId) )
  (PersonMove (location ?loc) (id ?staffId) )
  (Staff (id ?staffId) )
  (PersonMove (location ?loc) (id ?pttId) )
  (Patient (id ?pttId) )
  (ActivityBarProgram (id ?progid) (location ?loc))
  =>
  (sendDocumentMedicineActivity ( ?progid ?staffId ?pttId ) )
)
```

which corresponds to the activity that nurse Christensen enacts in the scenario section[1]. Based upon the presence of a medicine tray, the associated patient, and a nurse, we infer an activity to record that the patient has taken the medicine.

Thus, the activity discovery component combines person- and equipment move facts with facts from the electronic patient record system and heuristics about recurring work processes to infer likely healthcare activities. Here the power of the logical programming paradigm truly shows. As the above rule demonstrates, it is a fairly simple rule that encodes what would have been a complex and error-prone piece of code in a procedural language and correctly handles complex situations such as two or more nurses attending the same patient at the same time (both gets the opportunity to initiate the activity) and/or several running activity bar programs are present in the same location (the activity pops up on the activity bar on all computers in the given location allowing the nurse to choose which computer to use). Note that the latter case includes mobile computers, like PDAs, as a special case. In a procedural language the rule would be complex to write and it had to be iterated to account for all combinations of equipments and persons.

5.3 Handling Low Level Processing

Our implementation effort quickly showed that the low level processing in the location- and context server was also easily expressed as rules. Both location- and context server perform processing that basically transform data from lower- to higher levels of abstraction.

The location server is notified by the hardware level about 64bit tag IDs seen by a given tag-scanner and inserts a `TagEnter` fact with logical identities of tag and scanner as values into the knowledge base. Thus, the location server uses the knowledge base as a convenient database.

The context server in turn maintains facts about physical location of scanners and the identity of tags worn by persons and things like trays. Rules

[1] Time is not modeled in the present prototype.

retract `TagEnter` facts and replace them with appropriate `PersonMove` or `EquipmentMove` facts that describe the person/thing moving and its new physical location. Thus, rules are used to map from hardware events to events with high semantic contents.

5.4 Possible Future Activity Modeling

We have focused on medication and the nurses' activities in our project so far. Obviously many other types of activities can also be detected with high probability. Admitting a patient to a hospital means assigning him or her to a bed in the ward. Thus, if a nurse and a patient with undefined bed assignment happen to be near a bed, it is likely that the nurse is about to assign a bed to the patient. Nurses can maintain work lists where tasks may be triggered when a given location is visited or a given person is nearby. The graphical user interface of the EPR system may change based on work situation: for instance physicians use different setups for working at the ward, at doctors' conferences, and at the outpatient department—again our activity discovery component can infer the location of physicians and propose to change the EPR setup. If a physician is on the ward round and approaching the bed of a patient the ADC may trigger an activity in case new lab results have come in since she last visited the patient.

5.5 Metrics

As outlined in section 2, we have focused on functionality from the end-user perspective and have not considered architectural issues in depth. In our current prototype the knowledge base contains about 80–90 facts during our role-play scenarios, and we have about 70 rules. Thus, run-time performance, memory requirements, and response time have not been issues to worry about. These issues must of course be addressed if an activity-driven infrastructure is going to be deployed in a realistic setting. In a hospital like Aarhus county hospital, the knowledge base must be able to handle a large number of concrete objects. Aarhus county hospital has about 400 beds in 21 wards and about 1600 employees. In 2001 there were 19.200 admissions to the hospital, about 97.900 outpatient treatments, 29.800 consulted the casualty department, and 19.500 other types of treatments were made. That is about 450 patients per day. On top of that, we need objects/facts that model locations, computational devices and the activity bar programs running on them, and all interesting equipment like medicine trays, wheel chairs, beds, and other devices. Put together, it is obvious that a single, centralized, expert system must be able to cope with a large set of data.

Regarding the complexity of the rules, most of them are pretty straight forward as indicated in the examples described above. However, in some cases we need to control the order in which rules fire and therefore must introduce "pseudo-facts" whose only purpose is to guaranty the ordering.

6 Discussion

Pervasive computing is associated with "anywhere and anytime computing". Bringing computing to us in our everyday endeavors will change the way we perceive computers. The shift from mainframes to desktop computers changed the view from an *application-centered* to a *document-centered* perspective. We think that pervasive computing will once again change the perspective to a *human activity centered* perspective where our activities decide what information is relevant, how to present it, and what combination of equipment to use in order to manipulate it. Thus, our activity-driven computing infrastructure, and the rule-based approach, has wider applications than just healthcare.

The prototype was evaluated from two perspectives: from a functionality perspective and from a modifiability/maintenance perspective.

The functionality perspective was tested using scenarios at our evaluation workshop. These scenarios are small role-plays that take a well-known job situation as its starting point. The situation is rewritten to use the envisioned full-scale software solution and the clinicians are asked to "play" the situation using our prototype. The feedback from the clinicians was positive. Within the limited scope of handling medicine related activities, our activity discovery component was good at guessing relevant activities and the clinicians found the speed-up it gave them in handling the EPR system very important. In other situations, like prescribing medicine by physicians, activity guessing is more difficult as there are fewer rule-of-thumb on when it happens and fewer physical triggers like specific locations; for instance medicine is often prescribed for a patient in the corridor, not next to the patient's bed.

From the modifiability/maintenance perspective, we as programmers felt that using Jess gave us a number of benefits that were difficult to achieve otherwise. First, it gave us a declarative way of describing activities that is easier to write and maintain than corresponding procedural code. One exception, though, were in the few cases where the ordering of rule firing had to be controlled; here the programming is a bit tedious. Secondly, it gave us confidence that our programming was complete as the rule engine ensures that all possible combinations are tried. Thus, the benefit is both in short, easier to maintain, code as well as code with fewer errors. Third, the knowledge base became a common database of information shared by the location server, the context server, and the activity discovery component which also simplified programming: method invocations between the components as well as costly creation and subsequently garbage collection of objects are avoided and replaced by modifying shared facts.

Using an expert system is not without problems, however. One consideration we presently have is that of scalability. In our present infrastructure, rules are inferred over a single, centralized, knowledge base. In essence, the rule engine infers activities based on "global" knowledge. The question is how this scales to a realistic setting of a large hospital because of the large number of facts that must be dealt with in the knowledge base. Faster algorithms than RETE has been reported, like THREAT [21] but we do not find that speed or memory is the main issue: a more important concern is that the expert system becomes a *single*

point of failure. If it fails for some reason it will have hospital-wide consequences for the clinicians' work, which is problematic. One speculative idea is to abandon the idea of a global knowledge base in favor of a hierarchy of knowledge bases with local facts. For instance, we may have a knowledge base per ward that only maintains facts about that ward. If facts/events are interesting outside the ward itself, its knowledge base will inform its "parent" knowledge base using a *chain of responsibility* design pattern [17]. This way the failure of a knowledge base will only have local effects as well as the computational demands on memory- and processor speed are lessened.

A note from a programming point of view is that the programming model in Jess feels "flat" compared to our object-oriented programming model. As mentioned, three separate components share the knowledge base. These three components are implemented as classes in distinct Java packages ensuring encapsulation and information hiding as well as a hierarchical naming scheme. The Jess code for each component is also stored in distinct files but it is a weak modularity that only shows at the file level. At the Jess language level, all templates, facts, and rules are in a single, flat, name-space without information hiding. To object-oriented programmers, this seems primitive and we fear that the lack of proper scoping and encapsulation will make it difficult to maintain large amounts of Jess code. This problem also brings us doubts about the scalability of our approach. However, we acknowledge that there may be other expert system programming languages with better modularization support that we are not aware of.

7 Related Work

Our work relates to many aspects of research within pervasive and ubiquitous computing. Much research has focused on "intelligent environments". EasyLiving [6,13] explores the vision of an intelligent and, to some extent, activity-aware home. The activities are, however, much simpler than the ones encountered in healthcare, like for example "Tom logged into the desktop computer but has moved to the wall computer—thus move his computing session to the wall computer". This level of complexity is easily expressed in the procedural/object-oriented paradigm, and EasyLiving does not employ LP techniques for activity detection.

Many projects are concerned with environments and devices adapting to user context, notably user location, such as for example the exhibition guide Hippie [22], location based notification systems like ComMotion [20] and CybreMinder [12], and location-based composite devices [23]. The same argument goes here: the rules are too simple to require LP techniques.

Jaffar et al. describe an interesting healthcare system that also employs a LP component [19]. The system is also centered on patient treatment and medication. Basically, it is a workflow system where physicians' prescriptions generate a series of *work items* (similar to our notion of activities concerning giving medicine to patients and documenting it) that are inserted into the nurses' work

lists. Work items have an associated timestamp indicating when they must be initiated and completed. We have also considered the (unavoidable) issue of workflow—many activities are indeed organized with a natural ordering in time. They also use a LP component to generate work items/activities, however, the basis is the doctors' prescriptions, not the tracking of people and artifacts as in our case. Thus, their system is more rigid and focused, and the nurses' work schedule is more strongly dictated by a single artifact: the prescription. In contrast, our approach tries to help clinicians in whatever situation they may be in based on what they *are* doing, not what they were supposed to do according to a computer system.

8 Summary

We have described our experiences of using logic programming techniques within pervasive computing. Our research has been within the domain of healthcare and our objective has been to support everyday activities in healthcare to augment patient record data quality and, in particular, to ease and speed up the use of EPR systems. Our approach has been to design and experiment with an activity-driven computing infrastructure. Based on location-awareness and pervasive computing devices, the infrastructure is able to make qualified guesses about activities and propose these to the healthcare staff. Activities embody both EPR data and user interface setup and are inferred by an expert system. Declarative rules define activities in terms of the location of persons and things as well as heuristics about recurring work processes. This logic programming approach has many benefits compared to an imperative programming approach. However, we have also identified weaknesses in the approach, primarily concerning single-point-of-failure and scalability. The notion of a centralized knowledge base may not scale well for large organizations. The lack of language primitives in Jess for expressing modularity and information hiding also poses a scalability problem for the programming effort. Nevertheless, we find that expert systems have an important role to play in activity-centered and pervasive computing.

Acknowledgements

The activity-driven computing infrastructure was designed and implemented in collaboration with Jakob E. Bardram, Claus Bossen, and Anders K. Olsen. Anders K. Olsen contributed significantly by introducing Jess in the location- and context server components.

Thanks to the anonymous reviewers for valuable comments and especially to Maria Garcia de la Banda for providing guidance in preparing the final version of this paper.

References

1. J. E. Bardram. Scenario-based Design of Cooperative Systems: Re-designing an Hospital Information System in Denmark. In *Group Decision and Negotiation*, volume 9, pages 237–250. Kluwer Academic Publishers, 2000. 422
2. J. E. Bardram and C. Bossen. Interwoven Artifacts—Coordinating Distributed Collaboration in Medical Care. Submitted for "CSCW 2002" conference. 422
3. J. E. Bardram and H. B. Christensen. Supporting Pervasive Collaboration in Healthcare — An Activity-Driven Computing Infrastructure. Submitted for "CSCW 2002" conference. 422
4. J. E. Bardram and H. B. Christensen. Middleware for Pervasive Healthcare – A White Paper. In G. Banavar, editor, *Advanced Topic Workshop—Middleware for Mobile Computing*. http://www.cs.arizona.edu/mmc/Program.html, Heidelberg, Germany, Nov. 2001. 422
5. L. Brownston, R. Farrel, E. Kant, and N. Martin. *Programming Expert Systems in OPS5*. Addison Wesley, 1985. 427
6. B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. EasyLiving: Technologies for Intelligent Environments. In Thomas and Gellersen [25], pages 12–29. 433
7. J. M. Caroll. *Scenario Based Design: Envisioning work and technology in system development*. John Wiley & Sons, Inc, 1995. 422
8. Center for Pervasive Computing. www.pervasive.dk. 421
9. H. B. Christensen, J. Bardram, and S. Dittmer. Theme One: Administration and Documentation of Medicine. Report and Evaluation. Technical Report CfPC-2001-PB-1, Center for Pervasive Computing, Aarhus, Denmark, 2001. www.pervasive.dk/publications. 426
10. H. B. Christensen and J. E. Bardram. Supporting Human Activities — Exploring Activity-Centered Computing. Submitted to "UBICOMP" 2002 conference, 2002. 422
11. CLIPS: A Tool for Building Expert Systems. http://www.ghg.net/clips/CLIPS.html. 427
12. A. K. Dey and G. D. Abowd. CybreMinder: A Context-Aware System for Supporting Reminders. In Thomas and Gellersen [25], pages 172–186. 433
13. Easy living. http://research.microsoft.com/easyliving. 433
14. C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982. 429
15. E. Friedman-Hill. Jess, the Rule Engine for the Java Platform. http://herzberg.ca.sandia.gov/jess/. 427
16. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998. 429
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994. 433
18. M. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann Publishers, 1993. 427
19. J. Jaffar, M. J. Maher, and G. Neumann. An Architecture and Prototype Implementation of a System for Individualised Workflows in Medical Information Systems. In *Proceedings of the 23nd Hawaii International Conference on System Sciences*, 1999. 433
20. N. Marmasse and C. Schmandt. Location-Aware Information Delivery with ComMotion. In Thomas and Gellersen [25], pages 157–171. 433

21. D. P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 42–47, 1987. 429, 432
22. R. Oppermann and M. Specht. A Context-Sensitive Nomadic Exhibition Guide. In Thomas and Gellersen [25], pages 128–142. 433
23. T.-L. Pham, G. Schneider, and S. Goose. Exploiting Location-Based Composite Devices to Support and Facilitate Situated Ubiquitous Computing. In Thomas and Gellersen [25], pages 143–156. 433
24. Pervasive Healthcare. www.healthcare.pervasive.dk. 421
25. P. Thomas and H. W. Gellersen, editors. *Proceedings of Handheld and Ubiquitous Computing*, volume 1927 of *Lecture Notes in Computer Science*, Bristol, UK, sep 2000. Springer Verlag. 435, 436
26. M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7):75–84, July 1993. 421

Logic Programming for Software Engineering: A Second Chance

Kung-Kiu Lau[1] and Michel Vanden Bossche[2]

[1] Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk
[2] Mission Critical
Drève Richelle, 161 Bat. N, 1410 Waterloo, Belgium
mvb@missioncriticalit.com

Abstract. Current trends in Software Engineering and developments in Logic Programming lead us to believe that there will be an opportunity for Logic Programming to make a breakthrough in Software Engineering. In this paper, we explain how this has arisen, and justify our belief with a real-life application. Above all, we invite fellow workers to take up the challenge that the opportunity offers.

1 Introduction

It is fair to say that hitherto Logic Programming (LP) has hardly made any impact on Software Engineering (SE) in the real world. Indeed it is no exaggeration to say that LP has missed the SE boat big time! However, we have good reasons to believe that current trends in SE, together with developments in LP, are offering a second chance for LP to make a breakthrough in SE. In this application paper, we explain how this situation has arisen, and issue a "call to arms" to fellow LP workers in both industry and academia to take up the challenge and not miss the SE boat a second time!

2 The Past

Before we explain the current situation in SE, it is instructive to take a brief retrospective look at both SE and LP.

2.1 SE: The Software Crisis

SE has been plagued by the *software crisis* even before the term was coined at the 1968 NATO Conference on Software Engineering at Garmisch. Despite progress from structured or modular to object-oriented methodologies, the crisis persists today. As a result, software is not trusted by its users. At the European Commission workshop on Information Society Technologies, 23 May 2000, an invited expert from a major microelectronics company stated that "major advances in

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 437–451, 2002.
© Springer-Verlag Berlin Heidelberg 2002

microelectronics increase the pressure on software, but the fundamental problem is that we don't trust software".

So-called Formal Methods, e.g. VDM [15], Z [26] and B [1], were introduced to address the issue of software correctness. Whilst these have been successfully applied to several safety, were introduced to address the issue of software correctness. Whilst these have been successfully applied to several safety-critical projects, their practical applicability has been limited due to the high cost they incur. Additionally, there is the problem of "impedance mismatch" between a mathematical specification and an implementation based on traditional imperative languages such as C, Ada, etc.

2.2 LP: Unexplored Potential for SE

Like any declarative language, LP languages like Prolog can offer much to alleviate the software crisis. In particular, they can address software *correctness*.

A theoretically sound declarative language allows:

(i) the construction of a purely logical/functional version of the program based on a clear declarative semantics; and
(ii) the transformation into an efficient program.

With commonly used programming languages, correctness is hard to obtain (and to prove) whereas high-level declarative languages support and nurture correctness.

Software correctness, or the lack of it, is of course at the heart of the software crisis. So LP would seem to have the potential to make an important contribution to alleviating the software crisis. However, in the past, Prolog (or any other declarative language) was never seriously applied to SE in industry. This may be due to various factors, e.g. it may be because Prolog did not have the necessary features for programming-in-the-large, or it maybe because Prolog, or even the whole LP community, was not motivated by SE, and so on. Anyway, whatever the reasons (or circumstances), the consequence was that the potential of LP for SE has not been properly explored hitherto.

2.3 SE and LP: The Integration Barrier

Not even the staunchest LP supporter would claim that LP could compete on equal terms with the traditional imperative paradigm, especially OO Programming (OOP), for SE applications in general. So it is not realistic to expect LP to take over completely from the imperative paradigm that is predominant in SE. Rather, the only realistic goal is for LP to co-exist alongside the latter.

We believe LP's role in this co-existence is to address the critical kernel of a software system, for which there is no doubt that LP would be superior (for the reason that LP can deliver software correctness, as explained in Section 2.2). It is generally accepted that the critical kernel of a software system usually consists of some 20% of the code (see Figure 1), and it is this code that needs a scientific

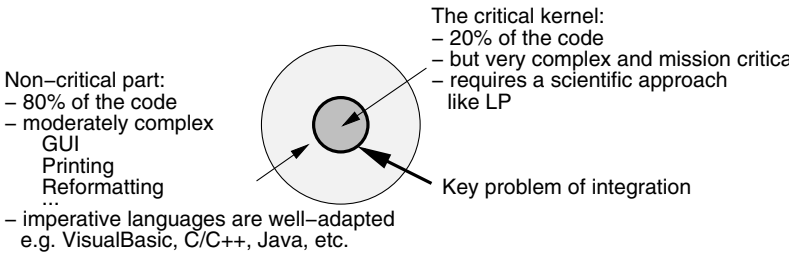


Fig. 1. The integration barrier between LP and predominant paradigms in SE

approach such as LP affords. However, even if LP was used for the critical code, the problem of integrating (critical) LP code with (non-critical) code in the predominant (imperative) languages would at best be difficult. For example, we could use a foreign function interface, usually in C, but this is often difficult. Thus, as also shown in Figure 1, there is an integration barrier between LP and the predominant paradigm in SE. This barrier would have to be overcome even if we were to use LP just for the critical kernel.

3 The Present

The software crisis persists today, despite the 'OO revolution'. LP has still not made any impact on SE. However, both areas show portentous movements.

3.1 SE: Dominated by Maintenance

Current industrial programming paradigms lack the sound and reliable formal basis necessary for tackling the inherent (and rapidly increasing) complexity of software, the extraordinary variability of the problem domains and the continuous pressure for changes. Consequently, current SE practice and cost are dominated by maintenance [11]. This is borne out by the many studies, e.g. [2], that strongly suggest that around 80% of all money spent on software goes into maintenance, of which 50% is corrective maintenance, and 50% adaptive (improvements based on changed behaviour) and perfective maintenance (improvements based on unchanged behaviour) [11]. This is illustrated in Figure 2 (taken from [11]).

3.2 SE: Moving to Components

Of course if software was more reliable, then the maintenance cost would decrease. Reuse of (reliable) software would reduce production cost. However, the level of reliability and reuse that has been achieved so far by the predominant OO approach is not significant. Large-scale reuse is still an elusive goal. It is therefore not surprising that today, with the Internet, and rapid advances in distributed

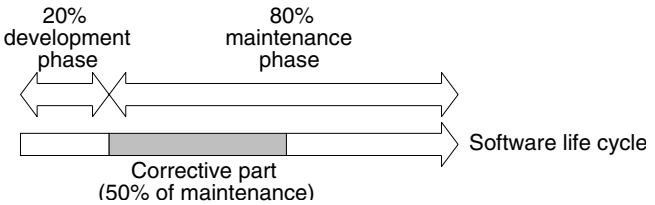


Fig. 2. Software cost is dominated by maintenance

technology, SE is seeking to undergo a 'paradigm shift' to Component-based Software Development (CBD).

Building on the concepts of Object-Oriented Software Construction (e.g. [22]), CBD [28] aims to move SE into an 'industrial age', whereby software can be assembled from components, in the manner that hardware systems are constructed from kits of parts nowadays.

The ultimate goal of CBD is thus third-party assembly of independently produced software components. The assembly must be possible on any platform, which of course means that the components must be platform-independent.

The consequences are:

A Level Playing Field. CBD offers a level playing field to all paradigms. Current approaches to industrial SE cannot address all the needs of CBD, so the playing field is level and LP is not at any disadvantage.

A Fast Developing Component Technology. Component technology for supporting CBD is receiving a lot of industrial investment and is therefore developing fast. The technology at present consists of the three component standards CORBA [10,3], COM [8] and EJB [27] supported by OMG, Microsoft and Sun respectively. Since by definition, it has to be platform and paradigm independent, this technology supports the level playing field.

These imply that CBD will overcome the integration barrier between LP and the predominant paradigm for SE, depicted in Figure 1. Thus CBD provides a realistic chance, for the first time, for LP to make a breakthrough in SE. We believe the importance of this cannot be overstated, and will devote a section (Section 3.4) to it.

3.3 LP: A Maturing Paradigm

In the meantime, LP has been maturing, as a paradigm for software development. Over the last ten years or so, the LOPSTR workshop series [19] has focused on program development. A theoretical framework has begun to emerge for the whole development process, and even tools have been implemented for analysis, verification and specialisation (see e.g. [6]).

A new logic-functional programming language, Mercury [25], has emerged that addresses the problems of large-scale program development, allowing mod-

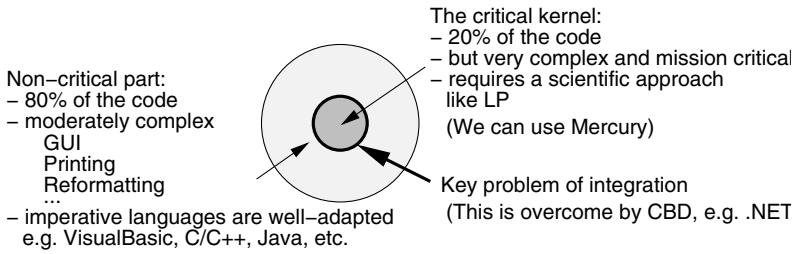


Fig. 3. CBD overcomes the integration barrier between LP and SE

ularity, separate compilation and numerous optimisation/time tradeoffs. It combines the clarity and expressiveness of declarative programming with advanced static analysis and error detection features. Furthermore, its highly optimised execution algorithm delivers efficiency close to conventional programming systems.[1]

So LP is in a good shape to take on the role of providing the 20% critical software as depicted in Figure 1.

3.4 SE and LP: CBD Overcomes the Integration Barrier

To reiterate, the crucial consequence of CBD, from LP's viewpoint, as mentioned in Section 3.2, is that component technology overcomes the integration barrier between LP and the predominant paradigm in SE. Therefore, we can update Figure 1 to Figure 3. This provides a realistic chance of a breakthrough for LP in SE.

We believe that a feasible practicable approach is to interface a suitable LP language, such as Mercury, to a current component technology. For example, we think that .NET [23], Microsoft's new component platform, could give LP the necessary component technology. As any language on a .NET platform can seamlessly interoperate with any other language on .NET (at least at a very low level), we have, for the first time, the possibility to build the critical components using the LP paradigm, while the non-critical, more mundane, components are still OOP-based.

This belief has propelled us at Mission Critical to invest in the "industrialisation" of Mercury [25], by interfacing it to .NET.

More specifically, we are working with the University of Melbourne on the following:

- integration with imperative languages through COM [8];
- multi-threading support [24];
- support for structure reuse, i.e. garbage collection at compile-time [20]: between 25% and 50% structure reuse has been observed in real-life programs;

[1] It is not our intention to engage in a 'language war' between Prolog and Mercury, or to debate our choice of Mercury.

- development of a suitable methodology which can guide developers who are confronted with a new programming paradigm;
- construction of a full .NET version of the Mercury compiler [9].

We have built a test Mercury.NET web service: a Coloured Petri Net component. Performance is very good, sometimes better than C#. There are still nitty gritty issues to solve (e.g. easier ways to produce the metadata related to an assembly), but they are being dealt with.

4 A Real-Life SE Application Using LP

The results of the "industrialisation" of Mercury have enabled Mission Critical to successfully develop a real-life system, part of which uses Mercury. The system was developed for FOREM, the regional unemployment agency of Wallonia (in Belgium). FOREM (with 3000 staff and an annual budget of 250 million euros) is confronted with complex and changing regulations, which directly impact many of its business processes. After several contractors had failed to develop a satisfactory system capable of supporting a new employment programme, FOREM asked Mission Critical to develop such a system.

The requirements for the system were as follows:

- it should have a 3-tier architecture with a clean separation between the User Interface, the Business Logic and the Data Storage;
- it should be Internet-ready, i.e. it should have good performance and robust security when the user interacts with the services through the Internet or the FOREM intranet;
- it should allow easy modification of the business processes to cope with a continuously changing regulation.

Mission Critical successfully developed a system that met these requirements. The system, PFlow (Figure 4), is in fact the first ever industrial Mercury application. It has been in daily use since September 2000.

4.1 System Architecture

PFlow is based on the following design and implementation:

- business process modelling is based on extended Petri Nets (to leverage their formal semantics, graphical nature, expressiveness, vendor independence, etc.);
- data modelling is ontology driven;
- the client/server protocol is based on XML and the WfMC (Workflow Management Coalition) XML Bindings recommendations;
- a light client is developed in Java;
- a complex server is developed in Mercury;
- business calculations based on Excel worksheets driven by the Mercury application;

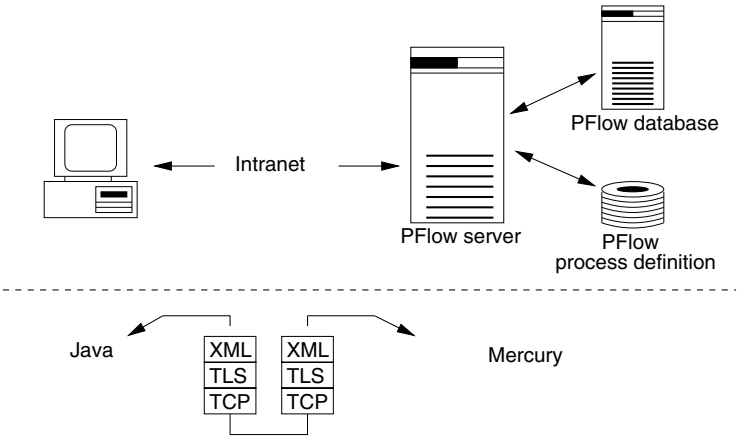


Fig. 4. Mission Critical's PFlow system

– component integration is done through COM.

The system architecture of PFlow is as shown in Figure 4. The main components of the system are a light client implemented in Java and a complex server developed in Mercury. The light Java client deals only with presentation issues, whereas the complex Mercury server provides the critical kernel of the system, providing a whole host of services including a Petri Net engine, folder management, alarms, business calculations, e-mail generation, transactions and persistence.

All state information in a Mercury program is threaded throughout the program as either predicate or function attribute values. To simplify the state information handling within the server a single structure, called *pstate*, encapsulates all relevant server state information. In the *pstate* structure, a general distinction has been made between values set at startup time (e.g. database names or SMTP server name and port number) and dynamic values which are constantly updated (e.g. folder cache, database connections). This distinction between the static and the dynamic state information means that options or customisation features can be added to modify the action of a part of the server with minimal effect on the other parts of the server.

The PFlow server operates in continuous loop accepting and processing requests as they arrive. When there are no outstanding client requests then the PFlow server uses the time to process any outstanding expired alarms, collect garbage, manage the cache and perform other internal housekeeping operations.

The protocol used between the clients and the PFlow server is a variant of the WfMC (Workflow Management Coalition) XML protocol. This protocol consists of several XML messages that must be sent to the server in a specific order (i.e. it is a stateful protocol) and the server must check the ordering. This protocol contains messages for creating a folder or a new task on a folder, finding tasks and updating a folder (and therefore its tasks, alarms, and dictionary entries).

The message sequence is divided into two parts: *task identification* and specific *resource querying* and *updating*. Tasks are defined as a part of the process description (currently there are 12 main tasks, each of which might have a number of sub-actions or sub-tasks which have to be complete before they are removed from the list). The XML messages in the sequence required by the server are:

- `PropFind`, used to recover client initialisation information;
- `Create-Task`, used to indicate to the server that the client is interested in a specific task;
- `PropFind-Folder`, to search for a folder (or list of folders);
- `Create-Folder`, to begin the folder updating;
- `PropFind-Folder`, to recover the folder based on its identifier;
- `PropPatch-Folder`, to update a folder data item, e.g. an alarm, dictionary entry or task value;
- `Terminate-Folder`, to end the folder updating;
- `TerminateTask`, to close the current task.

When a client starts up, the first request is a `PropFind`. on the dictionary definition, and this is also used to clear any database locks or other information which might be associated with that client (so in the event of a client computer crash or untidy exit, the client can always be restarted).

4.2 System Evaluation

Profiling the server indicates that one third of the message processing time is spent in DBMS related operations. This means that optimisations in the user time almost get partly overshadowed by the database access and update operations. Additional strategies are being investigated to tackle this problem and reduce the impact.

Another problem has been the use of Excel as the 'business computing engine'. In a first version of the system, the server called Excel directly through the COM interface. The response time (∼1sec to load the worksheet, send case data and retrieve derived value), although adequate for a prototype, was barely acceptable for a production system.

So, it was decided that the Mercury server would read the worksheet definition and use an internal Mercury representation. It would then interpret the Excel formulas internally (in Mercury) and keep the results as a Mercury data structure. This approach improved the response time considerably, reducing it to 30 msec (on a Pentium II, 350 MHz, 512 MB RAM), while keeping the standard Excel representation.

The current PFlow server has been used, in pilot mode, since March 2000, and in a full production environment, since September 2000, with currently 30 relatively intensive users across 3 sites. Since then the process description has evolved and been refined as requested by the customer. The system is being scaled up to 100 users working across 13 sites.

With the given number of users and sites, there are currently no performance problems and indeed quite the opposite since the work is being processed much

faster – and much more reliably – using the PFlow system than with the paper based approach. However, there are known areas where throughput can be improved and bottlenecks eliminated, for example, making portions of the server multi-threaded or distributing the work across several computers, if demanded by further workload increases.

4.3 Appraisal of Mercury

Deploying Mercury in the PFlow system has re-affirmed Mercury's strengths for system development in general. The strict declarative semantics of Mercury means that side-effect based programming is not easily possible, so hidden program assumptions are obvious during development and the subsequent maintenance. Moreover, the combination of a strong type and mode analysis and module system with a declarative reading means a virtual elimination of certain classes of typical programming and development problems (e.g. memory access problems, incorrect function/predicate attributes, wrong types). Eliminating these problems means that the majority of the development time is spent where it should be – in solving the more interesting higher level conceptual problems, such as when to recompute an alarm or when to commit to the database any folder updates.

Furthermore, in Mercury, correct program development from specifications is simpler, and therefore less time-consuming, than in non-declarative languages. This is because in Mercury there is no 'semantic gap' between a logic specification and its implementation.[2] In our experience this has definitely been the case. However, the need for efficiency may necessitate transforming the simplest possible (and obviously correct) program to one that is not quite so simple but more efficient (and less obviously correct). Even here, any transformation technique employed, e.g. co-routining,[3] must preserve the declarative semantics, i.e. maintain the 'no semantic gap' scenario.

Of course in general there are classes of problems where a'semantic gap' does exist, in the sense that Mercury may not be appropriate at all. For example, for constraint-solving problems, a constraint logic programming language would be more appropriate.

Moving on to performance, for PFlow, Mercury has also delivered. The Java client in PFlow, although only dealing with presentation issues, requires 22,000 lines of Java code, whereas the Mercury server is developed with no more than 18,000 lines of Mercury code. This bodes well for Mercury's performance as far as cost (both production and maintenance) and reliability are concerned, by any accepted criteria, e.g. those in [11].

[2] For one thing, negation in Mercury is *always* sound because we have full instantiatedness information, so we never try and negate a goal that is not fully ground.

[3] Co-routining can be implemented in Mercury, but the programmer has to do it explicitly using the concurrency library, modelled on Concurrent Haskell [24].

4.4 Supporting Evidence for LP for SE

The success of this application supports our view that LP can be used for the 20% critical software, and more importantly that LP can be integrated with the predominant paradigm in SE by using CBD technology. Indeed, the language features of Mercury make it a superior choice for implementing the critical kernel of the FOREM application, the Petri Net engine. Our implementation of this engine illustrates this point.

We implemented the Petri Net engine so that it supports coloured Petri Nets, and will execute as a component on the .NET backend. Coloured Petri Nets [14] are typed. Each place in the Petri Net can only contain tokens with a specified type, as well as the arc expressions which determine what tokens are placed into the places. A goal of the implementation was to use arbitrary Mercury functions for the arc expressions and allow tokens which are arbitrary Mercury types. The expressive Mercury type system, in particular existential types [12], ensured that Petri Nets can only be constructed in a type safe manner, eliminating one class of bugs completely.

The Petri Net state also has to be serialisable so that it could be persisted in a database, if necessary. The Mercury type system also allows this by ensuring that each type to be stored in a place must be a member of a typeclass [12,13] which can serialise and deserialise the type. Mercury's static type checking ensures that we can never construct a Petri Net which is not serialisable.

Finally Petri Nets are inherently non-deterministic. A transition fires when there exists a compatible token at each input place to a transition. This selection of tokens is modelled using committed choice non-determinism [21]. This allows the Mercury program to do the search to find the compatible tokens to consume, and then prune the choice point stack once a single solution is found. This gives us the benefits of the automatic search without paying the expense of the non-determinism once it is no longer needed.

5 The Future

We believe we are now at a critical juncture: our experience at Mission Critical has convinced us that LP has a chance to make a breakthrough in SE, but LP will only succeed if we collectively seize this opportunity in time.

5.1 SE: Component-Based Software Development

In the foreseeable future, SE will increasingly emphasise CBD. The move to CBD is seen by many as inexorable. As we have seen, CBD has opened up the possibility of LP's co-existence with older paradigms at the moment. In future, we believe that in addition to (mere) co-existence with other paradigms, LP can play a crucial role in CBD as a whole.

At present the key pre-requisites for meeting the CBD goal of third-party assembly have not been met (see e.g. [5]), these being: (a) a *standard semantics*

of components and component composition (and hence reuse); (b) good (*component*) *interface specifications*; and (c) good *assembly guide* for selecting the right components for building a specified system.

In [16,17,18] we argue and show that LP can play a crucial role in meeting these requirements. The cornerstone of our argument is that LP has a declarative semantics and that such a semantics is indispensable for meeting these prerequisites for CBD's success.

5.2 LP: Declarativeness Indispensable for CBD

So we believe that the role of LP in CBD is assured. The declarative nature of LP will increasingly come to the fore. As software gets more complex and networked, declarative concepts are increasingly recognised by industry as indispensable (see e.g. [4]). For example, declarative attributes are already common in security systems.

Industry are also beginning to realise and accept that declarativeness makes reasoning about the systems easier, and hence the systems are less likely to contain bugs. Even Microsoft is showing an interest in declarativeness. They have taken up the idea of expressing that a certain piece of executing code requires some constraints to be satisfied, and when these constraints are broken the system will refuse to execute. To reason about code containing constraints, of course you need a language with a simple semantics, e.g. a declarative one.

5.3 SE and LP: Predictable Software

Above all, the declarative nature of LP will enable it to be a key contributor to the ultimate goal for SE: *predictable software* built from *trusted components*.

In order for CBD to work, it is necessary to be able to reason about composition before it takes place. In other words, component assembly must be predictable; otherwise it will not be possible to have an assembly guide.

Consider Figure 5. Two components A and B each have their own interface and code. If the composition of A and B is C, can we determine or deduce the interface and code of C from those of A and B? The answer lies in component certification.

Component Certification. Certification should say what a component does (in terms of its *context dependencies*) and should guarantee that it will do precisely this (for all contexts where its dependencies are satisfied). A certified

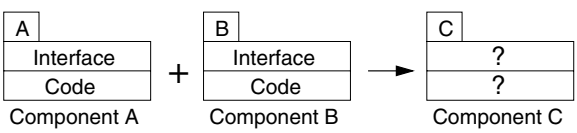


Fig. 5. Predicting component assembly

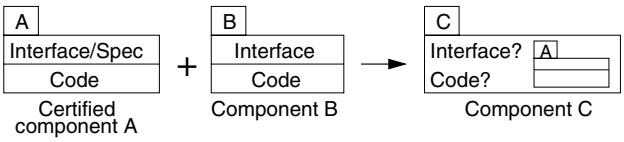


Fig. 6. Component Certification

component, i.e. its interface, should therefore be specified properly, and its code should be verified against its specification. Therefore, when using a certified component, we need only follow its interface. In contrast, we cannot trust the interface of an uncertified component, since it may not be specified properly and in any case we should not place any confidence in its code.

This is illustrated by Figure 6, where component A has been certified, so we know how it will behave in the composite C.

However, we do not know how B will behave in C, since it is not certified. Consequently, we cannot expect to know C's interface and code from those of A and B, i.e. we cannot predict the result of the assembly of A and B.

System Prediction. For system prediction, obviously we need *all* constituent components to be certified. Moreover, for any pair of certified components A and B whose composition yields C: (a) *before* putting A and B together, we need to know what C will be; (b) and furthermore, we need to be able to certify C.

This is illustrated by Figure 7. The specification of C must be predictable prior to composition. Moreover, we need to know how to certify C properly, and thus how to use C in subsequent composition.

As an example of predictable software, one of the most interesting (and difficult) subjects today is *secure* software systems, systems that can be trusted. This is becoming a very hot commercial issue, especially in the context of web services built by a company A that could consume other web services built by companies B, C, etc. To be trustworthy, these systems must be *predictable*. In order to trust a system, we must be able to predict that the software: (a) will do what it is expected to do; (b) will not do what could be harmful; (c) or should something bad happen, will detect this and regain control.

Finally, the issue of predictable software, or predictable component assembly, is becoming more and more important, and we believe that LP should be able to make a key contribution here (see [16,17,18] for a discussion).

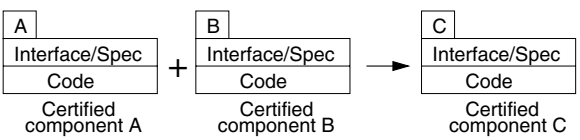


Fig. 7. System prediction

6 Discussion and Concluding Remarks

We have argued that CBD is giving LP a second chance to make an impact in real-world SE. Our belief stems from practical success of integrating LP to the traditional imperative paradigm via CBD technology, albeit using LP only for the critical kernel.

We have also stated our belief that in future, LP should be able to make a crucial contribution to the success of CBD as a whole. In particular, we believe that LP can be used to produce predictable software components.

Our sentiments here are very much echoed by voices on industrial forums such as CBDi Forum [7]:

> " ... the emphasis on well-formed components has diminished. This needs to be addressed and the necessity of good (trusted) component design communicated to all developers. ... "
>
> "Embrace formal component based specification and design approaches. Microsoft has already shown its interest in design by contract. This formal approach is a sensible basis for specification of trusted components. ... it is essential to understand and rigorously specify the behavior that the component or service, and its collaborations are required to adhere to. The conformance to behavioral specification is then a crucial part of a certification process which leads to trusted status.
> ... The challenge for Microsoft now is to provide support for delivery of trusted components and services, without reducing ease of use and productivity."

In addition, we also believe that logic and LP can be used for modelling and specifying software systems. The current standard of UML [**?**] has many limitations (not being formal enough), and we can do better than UML and have a completely formal logic-based modelling language.

Another interesting direction is "ontologies". The idea is to have a formal ontology describing problem domains. Logic and LP should be able to address the problem of the relation between the specifications and the ontology, the evolution of ontologies and specifications etc.

Finally, LP could aim at much more than a niche in SE. With the current rate of failures (Standish Group has observed that only 28% of projects are successful, i.e. 3 projects out of 4 have problems, and 1 in 4 is abandoned), a fundamental approach is needed. To borrow an engineering metaphor, you don't build a bridge with empiricism only (and debug it before you use it), you compute it first (with theories [mechanics], models [finite elements in the elastic domain], all "implemented" with mathematics). What we need is the same, i.e. the maths of software, discrete maths. LP is well grounded in this maths.

Acknowledgements

We are indebted to Peter Ross for his many helpful comments and points of information.

References

1. J. R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996. 438
2. R. S. Arnold. *On the Generation and Use of Quantitative Criteria for Assessing Software Maintenance Quality.* PhD thesis, University of Maryland, 1983. 439
3. BEA Systems *et al.* CORBA Components. Technical Report orbos/99-02-05, Object Management Group, 1999. 440
4. N. Benton. Pattern transfer: Bridging the gap between theory and practice. Invited talk at MathFIT Instructional Meeting an Recent Advances in Foundations for Concurrency, Imperial College, London, UK, 1998. 447
5. A. W. Brown and K. C. Wallnau. The current state of CBSE. *IEEE Software,* Sept/Oct 1998:37-46, 1998. 446
6. M. Bruynooghe and K.-K. Lau, editors. *Theory and Practice of Logic Programming.* Special issue an Program Development, 2002. 440
7. CBDi forum. http://www.cbdiforum.com. 449
8. The Component Object model Specification. Version 0.9, October 1995. http://www.microsoft.com/com/resources/comdocs.asp. 440, 441
9. T. Dowd, F. Henderson, and P. Ross. Compiling Mercury to the NET common language runtime. In *Proc. BABEL'01, Ist Int. Workshop an Multi-Language Infrastructure and Interoperability,* pages 70-85, 2001. 442
10. Object Management Group. The Common Object Request Broker: Architecture and specification Revision 2.2, February 1998. 440
11. L. Hatton. Does OO sync with how we think? *IEEE Software,* pages 46-54, May/Dune 1998. 439, 445
12. D. Jeffrey. *Expressive Type Systems for Logic Programming Languages.* PhD thesis, University of Melbourne, Submitted. 446
13. D. Jeffrey et al. Type classes in Mercury. Technical Report 98/13, Dept of Computer Science, University of Melbourne, 1998. 446
14. K. Jenses. A brief introduction to coloured petri nets. In *Proc. TACAS97,* 1997. 446
15. C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, second edition, 1990. 438
16. K.-K. Lau. The role of logic programming in next-generation component-based software development. In G. Gupta and I. V. R,amakrishnan, editors, *Proceedings of Workshop an Logic Programming and Software Enginering,* London, UK, July 2000. 447, 448
17. K.-K. Lau and M. Ornaghi. A formal approach to software component specification. In D. Giannakopoulou, G. T. Leavens, and M. Sitaraman, editors, *Proceedings of Specification and Verification of Component-based Systems Workshop at OOPSLA2001,* pages 88-96, 2001. Tampa, USA, October 2001. 447, 448
18. K.-K. Lau and M. Ornaghi. Logic for component-based software development. In A. Kakas and F. Sadri, editors, *Computational Logic: From Logic Programming into the Future.* Springer-Verlag, to appear. 447, 448
19. LOPSTR, home page. http://www.cs.man.ac.uk/-kung-kiu/lopstr/. 440
20. N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. In P. Codognet, editor, *Proc. 17th Int. Conf. an Logie Programming, LNCS 2237,* pages 105-119. Springer-Verlag, 2001. 441

21. Mercury reference manual. http://www.mercury.cs.mu.oz.au/information/documentation.html. 446
22. B. Meyer. *Objeet-oriented Software Construction.* Prentice-Hall, second edition, 1997. 440
23. Microsoft NET web page. http://www.microsoft.com/net/. 441
24. S. L. Peyton-Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23rd ACM Symposium an Prineiples of Programming Languages,* pages 295-308, 1996. 25. J. R,umbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999. 441, 445
25. Z. Somogyi, F. Henderson, and T. Conway. Mercury - an efficient, purely declarative logic programming language. In *Proc. Australian Computer Seience Comference,* pages 499-512, 1995. 440, 441
26. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992. 438
27. Sun Microsystems. Enterprise JavaBeans Specification. Version 2.0, 2001. 440
28. C. Szyperski. *Component Software: Beyond Objeet-Oriented Programming.* Addison-Wesley, 1998. 440

A Logic-Based System for Application Integration

Tamás Benkő, Péter Krauth, and Péter Szeredi

IQSOFT Intelligent Software Co. Ltd.
H-1135 Budapest, Csata u. 8.
{benko,krauthp,szeredi}@iqsoft.hu

Abstract. The paper introduces the SILK tool-set, a tool-set based on constraint logic programming techniques for the support of application integration. We focus on the Integrator component of SILK, which provides tools and techniques to support the process of model evolution: unification of the models of the information sources and their mapping onto the conceptual models of their user-groups.
We present the basic architecture of SILK and introduce the SILK Knowledge Base, which stores the meta-information describing the information sources. The SILK Knowledge Base can contain both object-oriented and ontology-based descriptions, annotated with constraints. The constraints can be used both for expressing the properties of the objects and for providing mappings between them. We give a brief introduction to SILan, the language for Knowledge Base presentation and maintenance. We describe the implementation status of SILK and give a simple example, which shows how constraints and constraint reasoning techniques can be used to support model evolution.

1 Introduction

The paper describes work on the SILK tool-set, carried out within the SILK project (System Integration via Logic and Knowledge) supported by the IST 5th Framework programme of the European Union, with the participation of IQSOFT Ltd. (Hungary, coordinator), EADS Systems & Defence Electronics (formerly Matra Systèmes & Information, France), the National Institute for Research and Development in Informatics (Roumania) and the Industrial Development and Education Centre (Greece).

The objective of the SILK project is to develop a knowledge management tool-set for the integration of heterogeneous information sources, using methods and tools based on constraints and logic programming.

The motivation of our work is the need to support the reuse of the valuable data handled by legacy systems (information sources), and the creation of composite application systems from them in order to solve new, higher level or wider scope business problems. Issues related to the evolution and maintenance of business oriented software systems are becoming the focus of work on application integration.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 452–466, 2002.
© Springer-Verlag Berlin Heidelberg 2002

SILK takes a fairly generic view of the information sources, supporting relational and object-oriented databases, semi-structured sources, such as XML files, as well as information sources accessible through programs or Web-services.

The SILK tool-set contains tools supporting both the dynamic access of heterogeneous information sources (mediation) and their transformation into a more homogeneous form (integration). Meta-data in SILK is represented in the form of object oriented *models* enhanced with *constraints*, and stored in a logic-based knowledge base. SILK includes tools for the verification of the models in the knowledge base, as well as for the comparison of models, uncovering their potential linking points and redundancies. This leads to more integrated models which are then used to support the inter-operability of underlying application systems and for their gradual transformation (evolution).

The paper is structured as follows. In Section 2 we introduce the varieties of meta-information handled by SILK. Section 3 describes the architecture of the SILK tool-set, while Section 4 presents the integration process of SILK. In Section 5 we introduce SILan, the SILK modelling language. Section 6 describes the SILK tools involved in the model evolution process and gives an example of this process. Section 7 gives a brief account on relationship between the SILK project and other ongoing research work, while Section 8 summarises our experiences regarding the use of logic programming and the construction of composite applications using SILK. Finally, Section 9 describes the further work planned in the SILK project, while Section 10 presents our conclusions.

2 Meta-information in SILK

Because of the central role of meta-information we discuss this issue first, before describing the SILK system architecture.

To carry out the task of integration, one has to build and maintain information on the sources to be integrated, on the way they can be linked, etc. This collection of meta-information is called the *SILK Knowledge Base* (KB). The SILK KB contains models, constraints and mappings, as the essential pieces of information used in the process of integration.

Models Models represent knowledge about structural properties of a system, i.e., about the entities and basic relationships between them. The notion of *model* in SILK is based on UML [17], with some extension from Description Logics [6].

Constraints It is often the case that we would like to reason about pieces of information which are not structural (for example, we would like to state a relationship between certain objects, e.g., that the concatenation of the first name and the last name yields the full name). We use *constraints* to describe such information.

Constraints in SILK are similar to those in OCL, the constraint sublanguage of UML. We use Constraint Logic Programming techniques [12] for reasoning with constraints.

Mappings If we want to perform queries involving multiple information sources, we have to build a *mapping* between their models, i.e., link the corresponding objects in the models, and describe their relationship (by a constraint).

The SILK tool-set handles models of two kinds: *application models* represent the structure of an existing or potential system and are therefore fairly elaborate and precise, while *conceptual models* represent mental models of user groups or domain knowledge encapsulated in ontologies, therefore they are vaguer than application models.

In another dimension, we can speak of *unified models*, i.e., models created from other ones in the process of integration and *local models*, e.g., specialised views of user groups or models of information systems (*interface models*).

To be able to elegantly express all the above types of models, SILK unites elements of the following important methodologies:

- object-oriented modelling for the description of structural properties of applications/systems;
- constraints for describing non-structural properties;
- description logics for the description of conceptual models.

The principal user of the SILK tools is the *knowledge engineer*, i.e., the person responsible for carrying out the process of application integration. End-users can access SILK either directly or indirectly through specific applications.

3 The SILK Architecture

The SILK system has three main subsystems as shown in Figure 1.

- The *Integrator* provides support for the knowledge engineer in building and using the SILK knowledge base. It contains tools for the entry and editing of models, tools supporting the creation of mappings and the unified models, and tools for restructuring of the information systems.
- The *Mediator* provides the means for the transformation of queries formulated in terms of conceptual models to queries of the underlying information sources [1].
- The *Wrapper* provides a uniform interface to the information sources, based on the object-oriented formalism. It supports handling of relational and object oriented databases, semi-structured formats such as XML, as well as information sources implemented as dedicated programs (e.g., web services). It also provides meta-information on the underlying sources, whenever possible.

The bulk of the Integrator and the Mediator components are implemented in Prolog. The Wrapper is implemented in Java to make use of the extensive library support for handling various data sources. The GUI of the Integrator is also implemented in Java. The Prolog and Java components communicate using

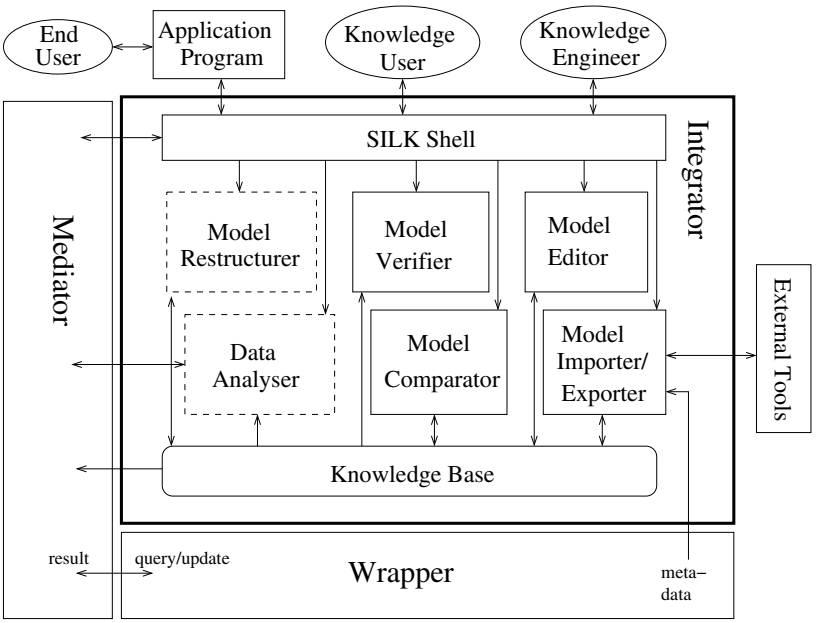


Fig. 1. Architecture of the SILK tool-set

Jasper, the Java connectivity library of SICStus Prolog [20], the implementation environment for the SILK tool-set.

In the following we shall focus on the Integrator component of the SILK tool-set. Accordingly, Figure 1 shows all sub-components of the Integrator. The components not yet implemented are shown with dashed lines. The fundamental element is the SILK Knowledge Base, this is the place where all models and mappings are stored. The SILK Shell provides both a graphical user interface and an application programming interface to all SILK components. The Model Editor is responsible for the maintenance of the KB. The task of the Model Comparator is to find similar model elements in two models. The Model Verifier can be used for checking the consistency of the models and mappings. The Model Restructurer provides support functions to create unified models. The Model Importer/Exporter can import and export models from external sources. The Data Analyser can be used to find inconsistencies and redundancies at the data level.

We describe the implementation of the main Integrator components in Section 6.

4 The SILK Integration Process

The most important phase of integration is the building up of new models in the knowledge base that are "better" than the original ones. There are at least two ways to improve the quality of a set of models: one is to bring them closer to the view of the users of the applications (which is the general principle of object-oriented modelling), another is to bring them closer to each other, i.e. to unify them. Unifying models (and, as a result, the applications themselves in a composite application) can be an improvement because we can eliminate redundancies and answer complex composite queries that involve many information sources.

Clearly, the integration process has to start from existing models. Such models can come from several places, from the information sources, from already existing models of sources prepared using external modelling tools (application layer models), or from the users (conceptual models). Usually these models correspond to different levels of abstraction with information source models being at the bottom and conceptual models at the top.

Obtaining models at the two lower levels is fully supported by SILK tools. Conceptual models can be created by using the model editing capabilities of the SILK Integrator, or indirectly by importing models from external sources.

Having obtained the initial models, we enter a loop of activities.

First, each individual model is checked for inconsistency and the knowledge engineer is prompted to resolve the contradictions uncovered.

Second, we have to find links between the models. For this purpose, the models are compared and tentative mappings are established between similar elements. If the models compared are on different levels of abstraction we talk about *abstractions*, if they are on the same level we call them *correspondences*. The two kinds of mappings are very similar but their use is different. Abstractions show how to bring our models closer to user views, while correspondences highlight redundancies and linking points between models.

Third, we introduce new models that unify existing models according to the correspondences between them.

Last, since the introduction of mappings and new models can give rise to new inconsistencies we enter the loop again. This series of activities is repeated until no inconsistency is found.

The Mediator component makes it possible to pose queries in the context of the newly developed models with the help of the mappings linking the models. If the results are satisfactory, the new models can be exported to standard modelling tools to serve as a basis for the development of a new system. When the new system is implemented, it can be filled with data from the original systems with the support of the SILK Mediator.

5 The Modelling Language SILan

The modelling language of SILK is called SILan. Its role is to support several knowledge base maintenance tasks, such as presentation and manipulation of

```
model Finance {
  class Employee {
    attribute String firstname;
    attribute String lastname;
    attribute Real salary, tax;
    constraint tax = 0.25*salary;
    constraint tax >= 10000;
    primary key (firstname,
                 lastname);
  };
};
```

```
model Production {
  class Worker {
    attribute String name;
    attribute String skills;
    attribute TimeTable timetable;
    attribute Real salary;
    constraint salary < 400;
    primary key name;

  };
};
```

Fig. 2. An example of the knowledge presentation syntax

models, describing information source capabilities, querying (mediation), etc. Therefore the SILan language has been designed to be expressive enough to describe models, information sources, queries, etc.

For almost all of the tasks described above there have already been defined some languages, many of which have been standardised as well. Rather than to design yet another language from scratch, we decided to re-use and unify the best features of the existing languages.

For the presentation and manipulation of models and other Knowledge Base elements we chose a syntax very similar to that of Corba IDL [15] (Interface Definition Language from OMG), which is in turn similar to ODL [4] (Object Definition Language from ODMG). This SILan sublanguage represents structural information, such as name-spaces (packages, models), classes, associations, etc, and provides a human readable interface to the KB for the knowledge engineer.

Constraints can be expressed in SILan using the OCL notation, with some enhancements. The additional language elements are necessary to provide a convenient syntax for constraints of Description Logics.

The syntax of data queries combines elements of the knowledge presentation, constraints and the query syntax of OQL (Object Query Language from ODMG).

In Figure 2 two (very simple) models are shown using the knowledge presentation language. Both models contain a single class. The classes have some attributes, some constraints, and primary keys. The primary key of the class `Employee` is compound. (The semantics of primary keys is the same as in database systems).

These two models may correspond to some information sources used by Finance and Production departments of a company. Both departments have a notion of a person working at the company, and use information sources for storing data about people. This is exemplified by classes `Employee` and `Worker`, some attributes of which are obviously related (such as `salary` and `...name`). In the next section we will show how these attributes are identified and linked.

6 Main Components of the SILK Integrator

In this section we describe those parts of the Integrator which have already been implemented: the SILK Knowledge Base, the Model Editor, the Model Importer/Exporter, the Model Comparator, and the Model Verifier. These components support the evolution of models, the mediation and a simple form of restructuring.

6.1 The SILK Knowledge Base

The heart of the SILK system is the Knowledge Base. This is the place where the models, the mappings, and other auxiliary information are stored. In this respect, the SILK KB can be considered as a Model Repository and the SILK Integrator as a model management facility.

The KB is implemented in three layers. The lowest layer is the physical storage of information. Above this we developed an API which hides the details of the physical implementation. This API is based on Prolog pattern matching. As a result, it is very high level and concise. The third layer is the SILan presentation format – this is produced by the Model Editor by repeated invocation of the KB access API.

The Knowledge Base API provides information on both the structural and logic aspects of the KB. The SILK components which are based on reasoning, namely the Model Verifier and the Mediator, require both kinds of information as first order clauses. Therefore we have developed an additional layer on top of the KB API, which converts the structural information to logic (e.g. inheritance is converted to an implication stating that each object of the derived class is also a member of the ancestor class). This logic API thus provides access to the KB in a uniform way, suitable for use in the reasoning components.

6.2 The SILK Shell and the Model Editor

The SILK Shell is the primary interface to the whole SILK tool-set. It is implemented as a graphical user interface with model browsing, model editing and querying capabilities. A console window providing access to the other components via a command shell is also embedded.

Browsing the Knowledge Base is facilitated by the Shell. Since the contents of the Knowledge Base form a graph, browsing it means looking at a given node (or a set of nodes). Every time during browsing there is a currently visited node, called *focus*. This focus serves as the default argument for the shell commands. (This is similar to browsing the directory structure of a file system.)

The Model Editor is closely connected to the Shell, since its most important task is to enable easy access to the contents of the Knowledge Base. The Editor uses the presentation format to display and read complete models or just simple model elements. Based on this, a simple editor window is provided, which displays the selected model element, allows its editing and replaces the original by the edited form.

6.3 The Model Importer/Exporter

The task of the Model Import/Export tool is to populate the Knowledge Base with the initial models and to support the export of the models resulting from the integration process to external modelling tools. Two main sources of external information are supported: the tool can read in UML models stored in XML files respecting the XMI specification as well as models of the actual data sources, if appropriate meta-information is provided by the source.

For handling XML files in Prolog we utilised the P*i*LL*o*W package [3].

6.4 The Model Comparator

The task of the Model Comparator is to find and connect similar elements in two sets of model elements. Mathematically, this involves the comparison of two graphs with many different kinds of nodes and leaves. Because of the freedom of modelling given by the object-oriented notation this task is very complicated and has a tendency to result in combinatorial explosion.

To cope with these problems we designed the Model Comparator to be configurable, modular, and interactive.

Almost all aspects of the Model Comparator can be configured by declarative descriptions. For example, we can choose to what extent different features of a model element influence its similarity if compared to some other element (not necessarily be of the same type). Such aspects of the comparison are weighted.

Modularity means that the Model Comparator is implemented as a set of so called *comparison methods*. A given method is responsible for the comparison of a given type of element. There is a method for comparing the nodes of the graphs as well as several methods for comparing different kinds of leaves, e.g., informal texts (comments), identifiers, etc.

Normally, the Model Comparator is used interactively and iteratively. The maximal recursion depth can be specified at each invocation, ensuring that an answer is found in a short time. Having obtained the result, it can be inspected using different *similarity thresholds*. This means that similarities with a weight less then the specified threshold will not be listed in the output. When an acceptable minimal similarity level is found, the knowledge engineer can confirm the good matches and discard the wrong ones. In the next step, he/she can request a new comparison either focusing on elements not yet compared or enabling the Model Comparator to go deeper in the recursion.

As the last step, the Model Comparator can introduce default mappings between the elements found similar. These have to be confirmed, completed, or corrected by the knowledge engineer.

6.5 The Model Verifier

The Model Verifier performs consistency checking of a set of model elements. It takes into account both structural information and constraints. Since our constraint language is equivalent in expressiveness to first order logic, the consistency of models is in general undecidable. To avoid infinite loops, the reasoning

process is limited both in depth and time. However, inconsistencies can be discovered in many practical cases, as exemplified by the next subsection.

When invoked with a set of model elements (classes and associations), the Model Verifier collects relevant parts of the Knowledge Base. These include both structural information and constraints and are translated to a uniform logical notation expressing both kinds of information. If this set of formulae is contradictory, one of the model elements in the set can have no instances and the set of elements is said to be inconsistent. Having found a contradiction, the verifier returns a locally minimal set of constraints which cause it, thus pinpointing the source of the problem.

Similarly to the Model Comparator, the Model Verifier is also implemented in a modular fashion. It has a central module called *scheduler* and several others called *solvers*. The task of the scheduler is to coordinate the invocation of the different solvers and to manage the pieces of information inferred by them. It is often the case that the inconsistency cannot be inferred by a single solver, but one of the solvers can deduce consequences of the known constraints which are subsequently used by another solver to find the inconsistency.

Currently, the Model Verifier uses the SICStus Prolog library CLP(R) [12] for reasoning on linear (in)equalities and the CHR [8] library for reasoning on strings, while the SetLog system {log} [5] is used for reasoning on sets.

6.6 An Example of Comparison and Verification

We conclude the section on Integrator components with an example showing the interaction of the Model Comparator and Model Verifier tools.

Let us consider the example of Figure 2, the two models describing the information sources used in two departments of a company. Let us assume that the models contain some further classes, associations, etc., in addition to the ones shown in the figure.

If we feed the two models to the Model Comparator tool, it will select the classes `Worker` and `Employee` as the ones corresponding to each other, based on the similarity of attribute names and types `(firstname,lastname)` ⇔ `name` and `salary` ⇔ `salary`. The Model Comparator will build a default mapping which will have to be refined by the knowledge engineer and checked by the Model Verifier.

Figure 3 presents the three phases of building a correct correspondence between the two classes, i.e. finding the constraint, which describes the proper relationship between the Worker and Employee classes.

The default correspondence, shown as the result of Step 1, is built by taking into account primary key information. Because the attributes `firstname`, `lastname` of the class `Employee` and the `name` of the class `Worker` are declared primary keys, the default mapping between the two classes is an implication saying that if `name` is equal to some unknown function of `firstname` and `lastname` then the two salaries should be equal.

Subsequently, the knowledge engineer replaces the unknown function with concatenation (let us assume that this is the correct function). The resulting

```
/* After Step 1: default correspondence */
correspondence (w: Production::Worker, e: Finance::Employee) {
   constraint w.name = unknown(e.firstname,e.lastname)
            implies w.salary = e.salary;
};

/* After Step 2: improved correspondence */
correspondence (w: Production::Worker, e: Finance::Employee) {
   constraint w.name = e.firstname.concat(e.lastname)
            implies w.salary = e.salary;
};

/* After Step 3: correct correspondence */
correspondence (w: Production::Worker, e: Finance::Employee) {
   constraint w.name = e.firstname.concat(e.lastname)
            implies w.salary*1000 = e.salary;
};
```

Fig. 3. Three phases of the introduction of a correspondence

correspondence, shown as the result of Step 2, is then checked by the Model Verifier. The verification finds an inconsistency in the mapping because of the constraints imposed on the salaries, and displays an arbitrarily chosen locally minimal set of contradictory constraints[1]:

```
e.tax = 0.25 * e.salary, e.tax >= 10000,
w.salary = e.salary, w.salary < 400
```

To resolve the contradiction, the knowledge engineer now consults the users of the two "systems", and finds out that the Production department stores the salary in units of thousands. Therefore the mapping is corrected, as shown as the result of Step 3 in Figure 3. The Model Verifier finds no inconsistency in the corrected mapping, and so the knowledge engineer can proceed with his task of model unification.

7 Comparison with Related Work

There are several completed and ongoing research projects which aim at linking heterogeneous information sources. The SIMS system [21] focuses primarily on mediation between various kinds of possibly distributed information. The InfoSleuth system [14] features an agent based distributed mediator, using domain specific ontologies. The approach used by the Observer project [13] emphasises the use of multiple, distributed ontologies organised in a hierarchical fashion.

[1] The mapping can only be satisfied if there are no elements linked by it, and this is considered a modelling error.

The OnToKnowledge project [18] set a very ambitious goal of creating the OIL language, to be used as the standard for describing and exchanging ontologies. OIL combines frame-based object descriptions with constraints expressed in a variant of Description Logic. The IBROW project [11] aims to develop an intelligent broker service for Web based information sources. Its recent result is the specification of UPML, a language for describing knowledge components. ICOM [7] is CASE tool based on the Entity-relationship model supporting constraints in Description Logic.

As seen from the above brief listing, most of the projects focus on describing ontologies and reason about these using some form of Description Logic. While acknowledging the importance of ontologies, we position the SILK tool-set somewhat differently: we aim to support the process of integration using standard object oriented methodologies, and allowing fairly general forms of constraints. Hence the SILK modelling language is based on UML, and reasoning uses constraint logic programming. Although there are tools for interpreting OCL constraints [2, 9], i.e., evaluating a constraint, given a concrete data-set, currently no other system supports data-independent reasoning on OCL-like constraints, as far as we know.

The SILan modelling language also includes constraint formats of Description Logic, and we plan to implement reasoning capabilities on these forms of constraints in the final phase of the development of the SILK tool-set.

8 Experiences

The first packaged versions of the tool-set have recently been produced (2002 Q1). Currently, the implementation of the Knowledge Base and the SILan language is considered finished, the Model Comparator and the Model Verifier are in prototype state. The source code of the Integrator consists of about 10000 lines of Java code and 34000 lines of Prolog code (both including comments).

In its current state, the SILK tool-set was successfully applied by the four SILK partners in four different domains: botany, microbiology, health-care and cloth manufacturing. The applications are fairly complex, the largest, the Botany application handles 20 models, of which 14 are directly imported from information sources. Figure 4 shows a screen-shot of the SILK Integrator showing a small part of the Botany application and some output of the Model Comparator. The development of this application took about 1 man month including the exploration of the domain.

During the development of the applications the Model Editor was used to create the unified models. We used the Model Comparator to establish mappings between model elements. In many cases the models compared were almost isomorphic and the Comparator could automatically find the correct mappings, only the exceptional cases (e.g., those needing some conversion) had to be handled by the Knowledge Engineer.

We now summarise our experiences of using logic programming tools and techniques in the SILK project.

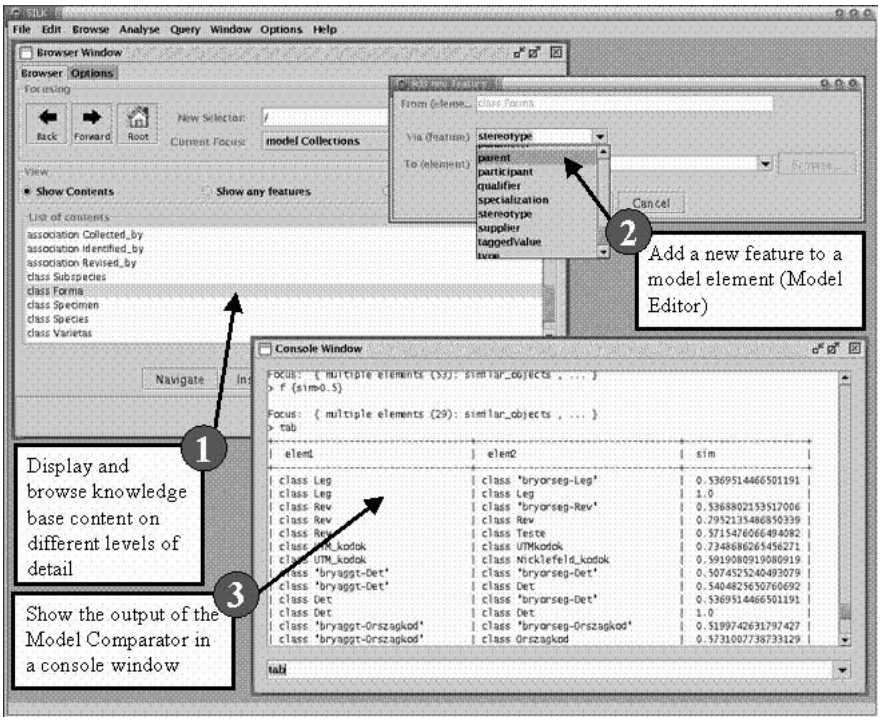


Fig. 4. A screen-shot of the SILK Integrator

As expected, Prolog proved to be very useful in parsing the SILan language. A tool was developed to transform a context free grammar to a Prolog parser, using the extended definite clause grammar formalism [19]. The same grammar, supplemented with formatting information, is also used for generating an "unparser", i.e., a program for producing the presentation format from the parse tree. The parser generator tool is also used for building parsers for the SILK Shell commands and the mediator queries.

We used the P*i*LL*o*W package [3] for the input and output of UML models in XMI format. The simple approach of building a character list from a file, and then using that for parsing, proved to be infeasible in case of larger models, both in terms of time and space. The time problem was overcome by implementing a C predicate for the input. The space issue was resolved by co-routining the input and the parsing processes and thus letting the garbage collector free the already processed part of the character list.

The SILK Knowledge Base API consists of just a few predicates, which take complex KB queries as arguments. The answer to the queries is returned by instantiating variables within the queries. This approach results in a very compact and general interface. The implementation of this API takes care of detecting

input-output patterns and using appropriate indexing techniques, ensuring adequate performance. While the initial implementation of the KB used the Prolog internal database, we now have switched to using a persistent database, based on the Berkeley DB [16] library of SICStus Prolog.

Constraint Logic Programming tools are used in Verifier and Mediator components. The Mediator uses CHR (Constraint Handling Rules) for transforming conceptual queries to those relating to the information sources [1]. The Verifier initially also used CHR, but by now it has been restructured to interface with several other solvers.

Finally, let us mention that the Jasper Java–Prolog interface of SICStus Prolog allowed a fairly smooth integration of the Java components, enabling us to access to the rich world of Java components.

All in all, Prolog and CLP proved to be invaluable in implementing the SILK tool-set within the tight resource and time constraints.

9 Future Work

Before improving the existing components and implementing the missing ones, the SILK tool-set will be evaluated and validated based on the experiences of the prototype applications.

Regarding the extension of existing components, an important task is to add new solvers to the Model Verifier (e.g., a solver for the constraints of Description Logics). We will also continue to tune the methods of the Model Comparator, based on the experiences of its application. Both of the above mentioned components will be made accessible through the graphical user interface.

We are also looking for ways to produce diagrammatic output of the Knowledge Base. At the moment two approaches seem to be feasible: either implement our own graphical components in Java or produce graphical descriptions in XMI format understood by many CASE tools.

There are two SILK components to be prototyped in the third, final phase of the project. The Model Restructurer will provide support for the refinement of models and will provide methods for transforming higher level models to lower level (application) models with a view to using the result in the refinement of the underlying information systems themselves. This process will also be supported by the Data Analyser, providing help for the system engineer in initialising a new system or transforming an existing one.

It is very promising to add support for various third party integration control tools (e.g., integration brokers) handling of control flow and supporting data synchronisation (eliminating redundancies) among applications in the composite application. We believe that the SILK knowledge based tool-set with strong reasoning capabilities will add significant value to the current application integration tools, and show the way for advanced automated support for such tools.

A further important research direction is to provide specific support for web-service integration, as the number of information sources of this kind is expected to grow rapidly.

10 Conclusion

We presented SILK, a tool-set for application integration based on logic. We described how we represent and handle structural and non-structural knowledge stored in the knowledge base. We illustrated the process of model refinement using the components of the SILK Integrator, and introduced SILan, a modelling language supporting the most prevalent standards in application development and database design.

We believe that the SILK tool-set will prove to be a very useful member of the growing family of tools supporting the object oriented modelling paradigm. We also believe that the choice of constraint logic programming, as the implementation technology, makes it possible to incorporate a wide range of reasoning techniques in the tool-set, which then can be used for both the analysis and for the mediation between information sources.

Acknowledgements

The authors acknowledge the support of the IST 5th Framework programme of the European Union for the SILK project (IST-1999-11135). We would like to thank all participants of the project, without whom the results presented in the paper would not have been possible to achieve. Special thanks are due to Attila Fokt for his work on the Model Comparator and the GUI, as well as to Imre Kilián for the development of the Model Verifier.

References

[1] L. Badea and D. Ţilivea. Query Planning for Intelligent Information Integration using Constraint Handling Rules, 2001. IJCAI-2001 Workshop on Modeling and Solving Problems with Constraints. 454, 464

[2] BoldSoft. Bold Architecture: Object Constraint Language. http://www.boldsoft.com/products/bold/ocl.html. 462

[3] D. Cabeza and M. Hermenegildo. WWW Programming using Computational Logic Systems. April 1997. Workshop on Logic Programming and the WWW. 459, 463

[4] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997. 457

[5] A. Dovier, E. Omodeo, E. Pontelli, and G. Rossi. {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1–44, 1996. 460

[6] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. C. A. Klein. OIL in a Nutshell. In *Knowledge Acquisition, Modeling and Management*, pages 1–16, 2000. 453

[7] E. Franconi. ICOM A Tool for Intelligent Conceptual Modelling. http://www.cs.man.ac.uk/ franconi/icom/. 462

[8] Th. Fruehwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Journal of Logic Programming*, volume 37(1–3), pages 95–138, October 1998. 460

[9] A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings 5th Asia Pacific Software Engineering Conference*. IEEE Computer Society, December 1998. 462

[10] I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. van Harmelen, M. Klein, S. Staab, R. Studer, and E. Motta. OIL: The Ontology Inference Layer. Technical Report IR-479, Vrije Universiteit Amsterdam, Faculty of Sciences, September 2000. See http://www.ontoknowledge.org/oil/.

[11] IBROW Project. An Itelligent Brokering Service for Knowledge-Component Reuse on the World Wide Web, 1999. http://kmi.open.ac.uk/projects/ibrow/. 462

[12] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP system. In J.L. Lassez, editor, *Logic Programming - Proceedings of the 4th International Conference*, volume 1. MIT Press, Cambridge, MA, 1987. 453, 460

[13] E. Mena, A. Illarramendi, V. Kashyap, and A. P. Sheth. Observer: An approach for query processing in global information systems based on interoperation across pre-existing ontologies, 1998. 461

[14] Microelectronics and Computer Technology Corporation. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments, 1997. 461

[15] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2, July 1995. 457

[16] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *1999 USENIX Annual Technical Conference, FREENIX Track*, pages 183–192, June 1999. 464

[17] OMG. Unified Modeling Language Specification, 1999. 453

[18] On-To-Knowledge Project. Tools for content-driven knowledge management through evolving ontologies, June 2000. http://www.ontoknowledge.org/. 462

[19] P. Van Roy. A Useful Extension to Prolog's Definite Clause Grammar Notation. *ACM SIGPLAN Notices*, 24(11):132–134, November 1989. 463

[20] SICS. *SICStus Prolog Manual*, April 2001. 455

[21] University of Southern California. SIMS Group Home Page. http://www.isi.edu/sims/sims-homepage.html. 461

The Limits of Horn Logic Programs*,**

Shilong Ma[1], Yuefei Sui[1], and Ke Xu[2]

[1] Department of Computer Science
Beijing University of Aeronautics and Astronautics, Beijing 100083, China
{slma,kexu}@nlsde.buaa.edu.cn
[2] Institute of Computing Technology, Academia Sinica, Beijing 100080, China
suiyyff@hotmail.com

It becomes more and more important to discover knowledge in massive information. The knowledge can be taken as a theory. As the information increases, the theories should be updated. Thus we get a sequence of theories, denoted by $\Pi_1, \Pi_2, \cdots, \Pi_n, \cdots$. This procedure may never stop, i.e. maybe there is not a natural number k such that $\Pi_k = \Pi_{k+1} = \cdots$. So sometimes we need to consider some kind of limit of theories and discover what kind of knowledge is true in the limit.

Li defined the limits of first order theories (Li, W., An Open Logic System, *Science in China* (Series A), 10(1992), 1103-1113). Given a sequence $\{\Pi_n\}$ of first order theories Π_n's, the limit $\Pi = \lim_{n\to\infty} \Pi_n$ is the set of the sentences such that every sentence in Π belongs to almost every Π_n, and every sentence in infinitely many Π_n's belongs to Π also. For a sequence $\{\Pi_n\}$ of finite Horn logic programs, if the limit Π of $\{\Pi_n\}$ exists then Π is a Horn logic program but it may be infinite. To discover what is true in Π, it is crucial to compute the least Herbrand model of Π. Then, the problem is: *How to construct the least Herbrand models of Π?* We know that for every finite Π_n, the least Herbrand model can be constructed. Therefore, one may naturally wonder if the least Herbrand model of Π can be approached by the sequence of the least Herbrand models of Π_n.

Let M_n and M be the least Herbrand models of Π_n and Π respectively. We hope to have $M = \lim_{n\to\infty} M_n$. It is proved that this property is not true in general but holds if under the assumption that *there is an N such that for every $n \geq N$, every clause $\pi : p \leftarrow p_1, \cdots, p_m$ in Π_n has the following property: for every i, if t is a term in p_i, then t is also in p.* This assumption can be syntactically checked and be satisfied by a class of Horn logic programs.

Thus, under this assumption we can approach the least Herbrand model of the limit Π by the sequence of the least Herbrand models of each finite program Π_n. We also prove that if a finite Horn logic program satisfies this assumption, then the least Herbrand model of this program is decidable. Finally, by the use of the concept of stability from dynamic systems, we prove that this assumption is exactly a sufficient condition to guarantee the stability of fixed points for Horn logic programs.

* The research is supported by the National 973 Project of China under the grant number G1999032701 and the National Science Foundation of China.

** The full paper is available at http://www.nlsde.buaa.edu.cn/~kexu.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 467, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Multi-adjoint Logic Programming: A Neural Net Approach*

Jesús Medina, Enrique Mérida-Casermeiro, and Manuel Ojeda-Aciego

Dept. Matemática Aplicada. Universidad de Málaga
{jmedina,merida,aciego}@ctima.uma.es

A neural implementation which provides an interesting massively parallel model for computing a fixed-point semantics of a program is introduced for multi-adjoint logic programming [3]. Distinctive features of this programming paradigm are that: very general aggregation connectives in the bodies are allowed; by considering different adjoint pairs, it is possible to use several implications in the rules.

A given multi-adjoint logic program P, its semantics is defined as the least fixpoint of an associated meaning operator (the immediate consequences operator T_P), which can be obtained by a bottom-up iteration from the least interpretation. The minimal model is implemented as a recurrent many-valued neural network where: (1) the confidence values of facts are the input values of the net; (2) confidence values of rules and the set of conjunctors, implications and aggregation operators in the bodies of the rules are used to determine the network functions; and (3) the output of the net gives the values of the propositional variables in the program under its minimal model (up to a prescribed approximation level). Regarding the structure of the net, each unit is associated to either a propositional symbol or an homogeneous rule (a standard clause for a fixed adjoint pair). Roughly speaking, we may consider three layers in the net: two of them are visible, representing the input and the output, respectively, and a hidden layer for the calculations with the homogeneous rules.

For the implementation we have considered the three main adjoint pairs in many-valued logic (product, Gödel and Łukasiewicz) together with a general operator of aggregation (interpreted as weighted sums). Nevertheless, it is an easy task to extend the model with new operators. As future work we are planning to relate our work with previous neural net implementations of logic programming, such as [1,2].

References

1. A. S. d'Avila Garcez and G. Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11(1):59–77, 1999. 468
2. S. Hölldobler, Y. Kalinke, and H.-P. Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11(1):45–58, 1999. 468
3. J. Medina, M. Ojeda-Aciego, and P. Vojtáš. Multi-adjoint logic programming with continuous semantics. In *Logic Programming and Non-Monotonic Reasoning, LPNMR'01*, pages 351–364. Lect. Notes in Artificial Intelligence 2173, 2001. 468

* Partially supported by Spanish DGI project BFM2000-1054-C02-02.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 468, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Fuzzy Prolog:
A Simple General Implementation Using CLP($\mathcal{R}$)*

Claudio Vaucheret[1], Sergio Guadarrama[1], and Susana Muñoz[2]

[1] Dept. de Inteligencia Artificial
claudio@clip.dia.fi.upm.es
sguada@isys.fi.upm.es
[2] Dept. de Lenguajes, Sists. de la Información e Ing. del Software
Universidad Politécnica de Madrid. 28660 Madrid, Spain
susana@fi.upm.es

Abstract. The result of introducing Fuzzy Logic into Logic Programming has been the development of several "Fuzzy Prolog" systems. These systems replace the inference mechanism of Prolog with a fuzzy variant which is able to handle partial truth as a real value or as an interval on $[0, 1]$. Most of these systems consider only one operator to propagate the truth value through the fuzzy rules.

We aim at defining a Fuzzy Prolog Language in a general way and to provide an implementation of a Fuzzy Prolog System for our general approach that is extraordinary simple thanks to the use of constraints.

Our approach is general in two aspects: (i) Truth value will be a countable union of sub-intervals on $[0, 1]$, representation also called Borel Algebra over this interval, $\mathcal{B}([0, 1])$. Former representations of truth value are particular cases of this definition and many real fuzzy problems only can be modeled using this representation. (ii) The concept of *aggregation* generalizes the computable operators. It subsumes conjunctive operators (triangular norms as min, prod, etc), disjunctive operators (triangular co-norms as max, sum, etc), average operators (arithmetic average, cuasi-linear average, etc) and hybrid operators (combinations of previous operators). We define and use *aggregation operator* for our language instead of limiting ourselves to a particular one. Therefore, we have implemented several *aggregation operators* and others can be added to the system with little effort.

We have incorporated uncertainty into a Prolog system in a simple way. This extension to Prolog is realized by interpreting fuzzy reasoning (truth values and the result of aggregations) as a set of constraints then translating fuzzy rules into CLP($\mathcal{R}$) clauses. The implementation is based on a syntactic expansion of the source code at compilation-time. The novelty of the Fuzzy Prolog presented is that it is implemented over Prolog, using its resolution system, instead of implementing a new resolution system such as other approaches. The current implementation is a syntactic extension that uses the CLP($\mathcal{R}$) system of Ciao Prolog. Lastest distributions includes our Fuzzy Prolog implementation and can be downloaded from http://www.clip.dia.fi.upm.es/Software. Our approach can be easily implemented on other CLP($\mathcal{R}$) system.

* Full version in http://www.clip.dia.fi.upm.es/papers/fuzzy-iclp2002.ps.gz

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 469, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Automated Analysis of CLP(FD) Program Execution Traces

Mireille Ducassé[1] and Ludovic Langevine[2]

[1] IRISA/INSA
Mireille.Ducasse@irisa.fr
http:/www.irisa.fr/lande/ducasse/
[2] INRIA Rocquencourt
Ludovic.Langevine@inria.fr

CLP(FD) programs can solve complex problems but they are difficult to develop and maintain. In particular, their operational behavior is not easy to understand. Execution tracers can give some insight of executions, but they are mapped onto the operational semantics of the language. This, in general, is a too low-level picture of the execution. In particular, application developers and end-users do not need to know all the details of the execution steps. They need abstract views of program behaviors.

Nevertheless, tracers and low-level traces are very useful in that they give a faithful view of the executions. We propose an approach where high-level views are built on top of low-level traces. An analysis module filters and tailors the execution traces produced by a tracer. A trace query language enables end-users to directly ask questions about executions. Application programmers can also provide explanation programs dedicated to the application. Solver developers can provide explanation programs dependent of their solvers. None of them need to know the implementation details of either the solver or the tracer.

Furthermore, we propose mechanisms to analyze the trace *on the fly*, without storing any information. For example, variable domains are tested as they appear in the data structures of the solver. The actual trace generation is directed by the analysis: the tracer only generates the information required by the analysis. It generates a big amount of information only if the analysis demands it.

The bases of the analysis scheme were initially designed for the analysis of Prolog programs [3]. We extended them to address the constraint store of CLP(FD), and we applied them to three constraint solving dedicated analyses. The first one gives a graphical view of the search tree. The second one gives a 3D Variable-Update View (similar to the one proposed by [2]). The third one gives an original view of the labeling procedure using the general graphical tool ESieve [1]. Each analysis is programmed in less than 100 lines.

This work is partially supported by the French RNTL project OADymPPaC, http://contraintes.inria.fr/OADymPPaC/.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 470–471, 2002.
© Springer-Verlag Berlin Heidelberg 2002

References

[1] T. Baudel. Visualisations compactes : une approche déclarative pour la visualisation d'information. In *Interaction Homme Machine*. ACM Press, 2002. 470

[2] M. Carro and M. Hermenegildo. The VIFID/TRIFID tool. Ch. 10 of, Deransart et al. editors, *Analysis and Visualization Tools for Constraint Programming*, *LNCS* 1870. Springer-Verlag, 2000. 470

[3] M. Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*, 39:177–223, 1999. 470

Schema-Based Transformations of Logic Programs in λProlog*

Petr Olmer and Petr Štěpánek

Department of Theoretical Computer Science and Mathematical Logic
Charles University, Prague
{petr.olmer,petr.stepanek}@mff.cuni.cz

This paper presents an application of higher-order logic programming: program schemata and schema-based transformations of logic programs. We are constructing higher-order programs that can unify logic programs with suitable program schemata, which are also higher-order constructs. They are abstracting out common recursive control flow patterns, and we can think about logic programs as about instances of certain program schemata. We use λProlog, because it can serve both as the language of logic programs and as the meta-language of program schemata.

Our schema-based transformation system in λProlog consists of the two phases: abstraction and specialisation. In abstraction, the logic programs we want to transform are abstracted to a set of program schemata. In specialisation, we apply transformations to this set: a chosen subset of program schemata is transformed and replaced by another output schema, and this process can be repeated to combine and compose transformations.

For us, a schema-based transformation defines a relation between program schemata: input ones, and an output one. A transformation is behavioural: the output program schema works in a different way from the input program schemata. A transformation is also structural: schema variables of the output program schema are instanciated differently from schema variables of the input program schemata.

We have developed transformations that introduce an accumulator (they create a tail-recursive programs), transformations that make use of unfold/fold rules (with other auxiliary transformations), and transformations that create B-stratifiable programs. We are running these transformations on program schemata processing lists and binary trees.

There is a need of manipulating formulas in the abstraction phase of transformation. An appropriate object logic gives an elegant solution to the problem. Structure of formulas we want to manipulate is expressed in term structures of λProlog and then processed. We define a toolset for creating and connecting our atomic formulas and terms, for managing a call of a program defined within the object logic, for managing λ-terms, and for manipulating a program schema as a whole.

The built transformation system is build as an open system; new transformations and program schemata can be added easily.

* The full version of the paper can be found at http://ktiml.ms.mff.cuni.cz/~olmer

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 472, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Non-uniform Hypothesis in Deductive Databases with Uncertainty

Yann Loyer and Umberto Straccia*

Istituto di Elaborazione della Informazione -C.N.R.
Via G. Moruzzi,1 I-56124 Pisa (PI) Italy

Many frameworks have been proposed for the management of uncertainty in logic programs (see, *e.g.* [1] for a list of references). Roughly, they can be classified into *annotation based* (AB) and *implication based* (IB) approaches. In the AB approach, a rule has form $A : f(\beta_1, \ldots, \beta_n) \leftarrow B_1 : \beta_1, \ldots, B_n : \beta_n$, asserting "the certainty of atom A is at least (or is in) $f(\beta_1, \ldots, \beta_n)$, whenever the certainty of atom B_i is at least (or is in) β_i, $1 \leq i \leq n$" (f is an n-ary computable function and β_i is either a constant or a variable ranging over an certainty domain). In the (IB) approach, a rule has form $A \overset{\alpha}{\leftarrow} B_1, ..., B_n$. Computationally, given an assignment v of certainties to the B_is, the certainty of A is computed by taking the "conjunction" of the certainties $v(B_i)$ and then somehow "propagating" it to the rule head. While the way implication is treated in the AB approach is closer to classical logic, the way rules are fired in the IB approach are more intuitive. Broadly, the IB is considered easier to use and more amenable for efficient implementation. Anyway, a common feature of both approaches is that the assumption made about the atoms whose logical values cannot be inferred is *equal* for all atoms: in the AB approach the *Open World Assumption* (OWA) is used (the default truth value of any atom is *unknown*), while in the IB approach this default value is the *bottom element* of a truth lattice, *e.g. false*. We believe that we should be able to associate to a logic program a semantics based on any given hypothesis, which represents our default or assumed knowledge. To this end, we extended, by providing syntax and a fixpoint semantics, the *parametric* IB framework [1] (an unifying umbrella for IB frameworks) along two directions: (i) we introduced *non-monotonic negation* into the programs; and (ii) an atom's truth may by default be either $\top$ (*e.g. true*), $\bot$ (*e.g. false*) or be *unknown*. A *rule* is of the form $r : A \overset{\alpha_r}{\leftarrow} B_1, ..., B_n, \neg C_1, ..., \neg C_m; \langle f_d, f_p, f_c \rangle$, in which f_d is a disjunction function associated with the predicate symbol A and, f_c and f_p are respectively a conjunction and a propagation functions associated with r. The intention is that the conjunction function determines the truth value of the conjunction of $B_1, ..., B_n, \neg C_1, ..., \neg C_m$, the propagation function determines how to "propagate" the truth value resulting from the evaluation of the body to the head, by taking into account the certainty α_r associated to the rule r, while the disjunction function dictates how to combine the certainties in case an atom is head of several rules. A *default assumption* is a partial function $H : B_P \rightarrow \{\top, \bot\}$, where B_P is the Herbrand base of a program P and $\top$ and $\bot$ are the top and bottom elements of a certainty lattice, respectively.

* Corresponding author: straccia@iei.pi.cnr.it

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 473–474, 2002.
© Springer-Verlag Berlin Heidelberg 2002

References

1. Laks V. S. Lakshmanan and Nematollaah Shiri. A parametric approach to deductive databases with uncertainty. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):554–570, 2001. 473

Probabilistic Finite Domains: A Brief Overview

Nicos Angelopoulos

Department of Computing, Imperial College
London, SW7 2BZ, UK
nicos@doc.ic.ac.uk

Abstract. We propose a new way of extending Logic Programming (LP) for reasoning with uncertainty. Probabilistic finite domains (*Pfd*) capitalise on ideas introduced by Constraint LP, on how to extend the reasoning capabilities of the LP engine. Unlike other approaches to the field, *Pfd* syntax can be intuitively related to the axioms defining Probability and to the underlying concepts of Probability Theory, (PT) such as sample space, events, and probability function.

Probabilistic variables are core computational units and have two parts. Firstly, a *finite domain*, which at each stage holds the collection of possible values that can be assigned to the variable, and secondly a *probabilistic function* that can be used to assign probabilities to the elements of the domain. The two constituents are kept in isolation from each other. There are two benefits in such an approach. Firstly, that propagation techniques from finite domains research are retained, since a domain's representation is not altered. Thus, a probabilistic variable continues to behave as a finite domain variable. Secondly, that the probabilistic function captures the probabilistic behaviour of the variable in a manner which is, to a large extent, independent of the particular domain values.

The notion of *events* as used in PT can be captured by LP predicates containing probabilistic variables and the derives operator ($\vdash$) as defined in LP. *Pfd* stores hold conditional constraints which are a computationally useful restriction of conditional probability from PT. Conditional constraints are defined by $D_1 : \pi_1 \oplus \ldots \oplus D_n : \pi_n \mid Q_1 \wedge \ldots \wedge Q_m$ where, D_i and Q_j are predicates and each π_i is a probability measure ($0 \leq \pi_i \leq 1$, $1 \leq i \leq n$, $1 \leq j \leq m$). The conjuction of Q_j's qualifies probabilistic knowledge about D_i. In particular, the constraint is evidence that the probability of D_i in the qualified cases (i.e. when $\vdash Q_1, \ldots, Q_m$) is equal to π_i. On the other hand a conditional provides no evidence for the cases where $\nvdash Q_1, \ldots, Q_m$.

Pfd has been used to model a well known example, the Monty Hall problem, which is often used to caution about the counter-intuitive results when reasoning with probabilities. Analysis of the computations over this model, has shown that *Pfd* emulates extensional methods that are used in statistics.

The main benefits of our approach are (i) minimal changes of the core LP paradigm, and (ii) clear and intuitive way for arriving at probabilistic statements. Intuitiveness of probabilistic computations is facilitated by, (a) separation of the finite domain and the probability assigning function of a variable, and (b) using predicates to represent composite events.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 475, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Modelling Multi-agent Reactive Systems

Prahladavaradan Sampath

Teamphone.com, London W1D 7EQ

Reactive systems are those that continuously interact with their environment asynchronously. A number of formalisms have been presented in the literature for reactive systems. However, these formalisms do not model systems that consist of a number of reactive sub-components dynamically interacting with each other and the environment. An example of this type of system can be seen in the telecommunications arena, where sophisticated telephony applications involve controlling a number of individual calls distributed across different switches, and where the interactions between the calls changes dynamically. Another example, is the modelling of multi-agent systems consisting of agents that are conceptually distinct from each other, and which maintain internal state and beliefs. The agents interact with each other and the environment, and can change their internal state and beliefs as they evolve over time.

We have developed a formalism for modelling systems consisting of reactive sub-components, that dynamically interact with each other and the environment. Our technique is based on extending an existing formalism for specifying reactive systems – Timed Concurrent Constraints – with a formalism for capturing the dynamic configuration of the reactive sub-components – the Ambient Calculus. We name the resulting formalism Mobile Timed Concurrent Constraints (MTCC). The central idea of the extension is that ambient names can be considered as *signals* in the Gentzen constraint system of TCC. The operational semantics of MTCC is presented as an extension of the semantics for TCC with alternating sequences of constraint evaluation and ambient reduction.

One of the motivations for using Timed Concurrent Constraints (TCC) over other formalisms in our work is the very simple and elegant semantic model presented in [2] for TCC as sets of sequences of quiescent states. The concepts introduced by the Ambient Calculus to model the multi-agent nature of systems are orthogonal to the concents in TCC, and the combination of the two gives an elegant model for multi-agent reactive systems.

We have developed an operational semantics for MTCC and are in the process of formalising a denotational semantics as an extension of the denotational semantics of TCC. Future work includes algorithms for compiling MTCC programs into automata, and developing logics for reasoning about MTCC agents.

References

1. P. Sampath. Modelling multi-agent reactive systems. Available from http://www.vaikuntam.dsl.pipex.com/reports.html.
2. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*. Paris, France, 1994. 476

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 476, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Integrating Planning, Action Execution, Knowledge Updates and Plan Modifications via Logic Programming

Hisashi Hayashi, Kenta Cho, and Akihiko Ohsuga

Computer and Network Systems Laboratory
Corporate Research and Development Center, TOSHIBA Corporation
1 Komukai, Toshiba-cho, Saiwai-ku, Kawasaki-shi, 212-8582, Japan
{hisashi3.hayashi, kenta.cho,akihiko.ohsuga}@toshiba.co.jp

Abstract. Prolog has been used as an inference engine of many systems, and it is natural to use Prolog as an inference engine of intelligent agent systems. However, Prolog assumes that a program does not change. This poses a problem because the agent might work in a dynamic environment where unexpected things can happen. In order to use a Prolog-like procedure as an inference engine of an agent, the procedure should be able to modify the computation, if necessary, after updating the program or executing an action. We introduce a new Prolog-like procedure which integrates planning, action execution, program updates, and plan modifications. Our new procedure computes plans by abduction. During or after a computation, it can update a program by adding a rule to the program or deleting a rule from the program. After updating the program, it modifies the computation, cuts invalid plans, and adds new valid plans. We use the technique of Dynamic SLDNF (DSLDNF) [1] [2] to modify computation after updating a program. It is also possible to execute an action during or after planning. We can use three types of actions: an action without a side effect; an action with a side effect which can be undone; an action with a side effect which cannot be undone. Following the result of action execution, the procedure modifies the computation: invalid plans are erased; some actions are undone; some redundant actions are erased. Even if a plan becomes invalid, it is possible to switch to another plan without loss of correctness. Based on the technique described above, we implemented an intelligent mobile network agent system, *picoPlangent*.

References

1. H. Hayashi. Replanning in Robotics by Dynamic SLDNF. IJCAI Workshop "Scheduling and Planning Meet Real-Time Monitoring in a Dynamic and Uncertain World", 1999. 477
2. H. Hayashi. Computing with Changing Logic Programs. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 2001. 477

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 477, 2002.
© Springer-Verlag Berlin Heidelberg 2002

A Logic Program Characterization of Domain Reduction Approximations in Finite Domain CSPs

Gérard Ferrand and Arnaud Lallouet

Université d'Orléans — LIFO
BP 6759 — F-45067 Orléans cedex 2
{Gerard.Ferrand,Arnaud.Lallouet}@lifo.univ-orleans.fr

Abstract. We provide here a declarative and model-theoretic characterization of the approximations computed by consistency during the resolution of finite domain constraint satisfaction problems.

Answer Set Programming [1] is a powerful knowledge representation mechanism in which logic program clauses are considered as constraints for their possible models. They have been used to model Constraint Satisfaction Problems (CSPs) in [2] and also in [1] and we contribute to this line of work by representing in this paradigm, extended to 3-valued logic, not only the solutions, but also the whole computational process. We propose to represent by a declarative semantics of a CLP program the approximation computed by a consistency.

We first propose a definite CLP program P and show that the greatest fixed-point of its associated operator T_P coincides with the consistency's computed approximation. This formulation enjoys also a nice logical reading since it is also the greatest model of the completed program P^*. But since the solving process does not end with the first consistency enforcement, we also show that consistent states obtained after arbitrary labeling steps are also modeled by downward closures of T_P starting from suitably restricted interpretations.

A second CLP program P_{neg} is obtained by a transformation of P by classical De Morgan laws and allows to model the individual contribution of each propagation rule. But since the program has negations, the semantics which turned out to be useful is the well-founded semantics [4]. Its negative part expresses the deleted values in the variable domains, the ones which do not participate to any solution. The computed approximation is thus completely defined by the complementary of the negative part of P_{neg}'s well-founded semantics. This allows to establish a deep link between traditional CSP solving methods based on consistency and the knowledge representation methods based on stable model semantics of logic programs since the well-founded semantics is the least 3-valued stable model [3].

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 478–479, 2002.
© Springer-Verlag Berlin Heidelberg 2002

References

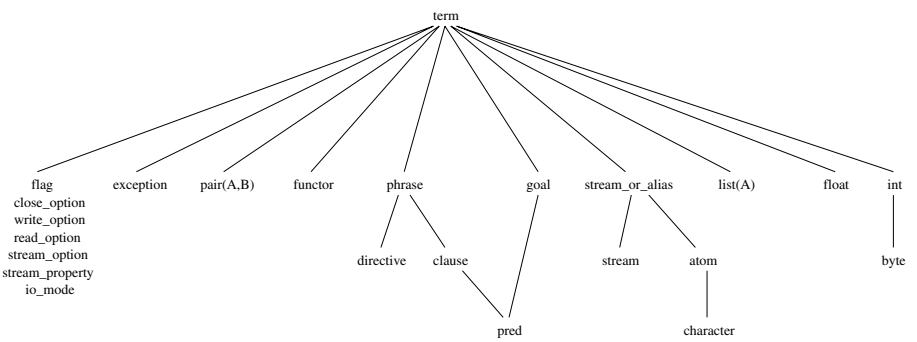
1. Victor W. Marek and Mirosław Truszczyński. *Stable Models and an Alternative Logic Programming Paradigm*, pages 375–378. Artificial Intelligence. Springer-Verlag, 1999. 478
2. Ilkka Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999. 478
3. T. Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, XIII:445–463, 1990. 478
4. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991. 478

TCLP: Overloading, Subtyping and Parametric Polymorphism Made Practical for CLP

Emmanuel Coquery and François Fages

Projet Contraintes, INRIA-Rocquencourt, France
{emmanuel.coquery,francois.fages}@inria.fr

This communication is a continuation of our previous work on the TCLP type system for constraint logic programming [1]. Here we introduce overloading in TCLP and describe a new implementation of TCLP in the Constraint Handling Rules language CHR. Overloading, that is assigning several types to symbols, e.g. for integer and floating point arithmetic, makes it possible to avoid subtype relations like integer subtype of float, that are not faithful to the behavior of some predicates. We describe a new implementation of TCLP in Prolog and CHR where overloading is resolved by backtracking with the Andorra principle. Experimental results show that the new implementation of TCLP in CHR outperforms the previous implementation in CAML [2] w.r.t. both runtime efficiency, thanks to simplifications by unification of type variables in CHR, and w.r.t. the percentile of exact types inferred by the TCLP type inference algorithm, thanks to overloading. The following figure depicts the TCLP type structure we propose for ISO Prolog. Metaprogramming predicates in ISO prolog basically impose that every object can be decomposed as a term. This is treated in TCLP by subtyping with a type *term* at the top of the lattice of types.



Type checking and type inference in TCLP have been evaluated on 20 Sicstus Prolog libraries including CLP libraries. The complete version of the paper is available at http://contraintes.inria.fr/~fages/Papers/CF02iclp.ps.

References

1. F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1, November 2001. 480
2. F. Pottier. Wallace: an efficient implementation of type inference with subtyping, February 2000. http://pauillac.inria.fr/~fpottier/wallace/. 480

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 480, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Logical Grammars Based on Constraint Handling Rules

Henning Christiansen

Roskilde University, Computer Science Dept.
P.O.Box 260, DK-4000 Roskilde, Denmark
henning@ruc.dk

CHR Grammars (CHRGs) are a grammar formalism that provides a constraint-solving approach to language analysis, built on top of Constraint Handling Rules in the same way as Definite Clause Grammars (DCGs) on Prolog. CHRGs work bottom-up and add the following features when compared with DCGs:

- An inherent treatment of ambiguity without backtracking.
- Robust parsing; do not give up in case of errors but return the recognized phrases.
- Flexibility to produce and consume arbitrary hypotheses making it straightforward to deal with abduction, integrity constraints, operators *á la* assumption grammars, and to incorporate other constraint solvers.
- References to left and right syntactic context; apply for disambiguation of simple and otherwise ambiguous grammars, coordination in natural language, and tagger-like grammar rules.

Example: The following rules are excerpt of a grammar for sentences with coordination such as *"Peter likes and Mary detests spinach"*. Complete sentences are described in the usual way and incomplete ones (followed by "`and` $\cdots$") take their subject from the sentence to the right:

```
          sub(A), verb(V), obj(B) <:> sent(s(A,V,B)).
subj(A), verb(V) /- [and], sent(s(_,_,B)) <:> sent(s(A,V,B)).
```

Marker "`/-`" separates the sequence "`subj-verb`" from its right context; "`<:>`" indicates a rule *á la* CHR's simplification rule. The following excerpt shows left and right context in action to classify nouns according to their position relative to the verb.

```
   n(A) /- verb(_) <:> subj(A).          verb(_) -~n(A) <:> obj(A).
n(A), [and], subj(B) <:> subj(A+B).  obj(A), [and], n(B) <:> obj(A+B).
```

CHRG also provides notation for propagation and simpagation rules. Examples with abduction and other features of CHRG are found at the web site below.

References

1. Christiansen, H., Abductive language interpretation as bottom-up deduction. To appear in: Wintner, S. (ed.), *Proc. of NLULP 2002, Natural Language Understanding and Logic Programming,* Copenhagen, Denmark, July 28th, 2002.
2. **Web site** for CHRG with source code written in SICStus Prolog, Users' Guide, sample grammars, and full version of the present paper:
   `http://www.ruc.dk/~henning/chrg/`

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, p. 481, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Debugging in A-Prolog: A Logical Approach

Mauricio Osorio, Juan Antonio Navarro, and José Arrazola

Universidad de las Américas, CENTIA
Sta. Catarina Mártir, Cholula, Puebla. 72820 México
{josorio,ma108907}@mail.udlap.mx

A-Prolog, Answer Set Programming or Stable Model Programming, is an important outcome of the theoretical work on Nonmonotonic Reasoning and AI applications of Logic Programming in the last 15 years. In the full version of this paper we study interesting applications of logic in the field of answer sets.

Two popular software implementations to compute answer sets, which are available and easy to find online, are DLV and SMODELS. Latest versions of these programs deal with disjunctive logic programs plus constraints. An important limitation is that, however, no tools for analyzing or debugging code have been provided. Sometimes, when computing models for a program, no answer sets are found while we were, in principle, expecting some of them to come out. We observed how an approach based on the three-valued logic G_3 can be useful to detect, for instance, constraints that are violated and invalidate all expected answer sets. These tools could help the user in finding the offending rules in the program and act in response correcting possible mistakes.

We introduce first the notion of *quantified knowledge* that is used to define an order among partial G_3 interpretations of logic programs. Then a notion of minimality between *implicitly-complete* interpretations, which can be uniquely extended to complete models, is defined in terms of this order. Such extended models are then called *minimal* models.

We defined then the *weak*-G_3 *semantics* as the set of minimal models for a given program, and the *strong*-G_3 *semantics* as the set of minimal models that are also definite (no atom is assigned to the undefined value of the G_3 logic). As a consequence of a characterization we provided for answer sets in terms of intermediate logics, we were able to prove that the strong-G_3 semantics corresponds exactly to the answer set semantics as defined for nested programs by Lifschitz, Tang and Turner [*Nested expressions in logic programs*, 1999].

The weak-G_3 semantics has, however, interesting properties we found useful for debugging purposes. Every consistent program has, for example, at least one minimal model and, since minimal models are not always definite, will detect atoms that are left indefinite. The intuition behind is that an answer set finder cannot decide, for these indefinite atoms, if they are either true or false and thus rejecting a possible model. It is also discussed how this ideas can be applied, using a simple transformation of constraints into normal rules, to detect violated constraints in programs.

P. J. Stuckey (Ed.): ICLP 2002, LNCS 2401, pp. 482–483, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Acknowledgements

This research is sponsored by the Mexican National Council of Science and Technology, CONACyT (project 35804-A). Full version of this paper available online at http://www.udlap.mx/~ma108907/iclp.

Author Index

Angelopoulos, Nicos 475
Antoniou, Grigoris 393
Arrazola, José 482

Banda, Maria García de la 38
Barker, Steve 54
Benkő, Tamás 452
Bockmayr, Alexander 85
Boigelot, Bernard 1
Bonatti, Piero A. 333
Bossche, Michel Vanden 437
Bruscoli, Paola 302
Bry, François 255

Cabalar, Pedro 378
Charatonik, Witold 115
Cho, Kenta 477
Christensen, Henrik Bærbak ... 421
Christiansen, Henning 481
Coquery, Emmanuel 480
Courtois, Arnaud 85
Craciunescu, Sorin 287

Decker, Stefan 20
Demoen, Bart 38, 179, 194
Dimopoulos, Yannis 363
Dobbie, Gillian 130
Ducassé, Mireille 470

Fages, François 480
Ferrand, Gérard 478

Ganzinger, Harald 209
Guadarrama, Sergio 469

Hayashi, Hisashi 477

Inoue, Katsumi 317

Jamil, Hasan M. 130

Kramer, Jeff 22
Krauth, Péter 452

Lallouet, Arnaud 478
Langevine, Ludovic 470
Lau, Kung-Kiu 437
Lonc, Zbigniew 347
Loyer, Yann 473

Ma, Shilong 467
Maher, Michael J. 148, 393
Makholm, Henning 163
Martin, Eric 239
McAllester, David 209
Medina, Jesús 468
Mérida-Casermeiro, Enrique ... 468
Miller, Rob 22
Mukhopadhyay, Supratik 115
Muñoz, Susana 469

Navarro, Juan Antonio 482
Nguyen, Phuong 239
Nguyen, Phuong-Lan 194
Nuseibeh, Bashar 22

Ohsuga, Akihiko 477
Ojeda-Aciego, Manuel 468
Olmer, Petr 472
Osorio, Mauricio 482

Pearce, David 405
Pemmasani, Giridhar 100
Pientka, Brigitte 271
Podelski, Andreas 115

Ramakrishnan, C. R. 100
Ramakrishnan, I. V. 100
Russo, Alessandra 22

Sagonas, Konstantinos 163
Sakama, Chiaki 317
Sampath, Prahladavaradan 476
Sarsakov, Vladimir 405
Schaffert, Sebastian 255
Schaub, Torsten 405
Schimpf, Joachim 224
Schrijvers, Tom 38
Sharma, Arun 239
Sideris, Andreas 363
Štěpánek, Petr 472

Stephan, Frank 239
Straccia, Umberto 473
Sui, Yuefei 467
Szeredi, Péter 452

Thielscher, Michael 70
Tompits, Hans 405
Truszczyński, Mirosław 347

Vandeginste, Ruben 194
Vaucheret, Claudio 469

Wolper, Pierre 1
Woltran, Stefan 405

Xu, Ke 467