Lecture Notes in Artificial Intelligence 2923

Edited by J. G. Carbonell and J. Siekmann

Subseries of Lecture Notes in Computer Science

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Vladimir Lifschitz Ilkka Niemelä (Eds.)

Logic Programming and Nonmonotonic Reasoning

7th International Conference, LPNMR 2004
Fort Lauderdale, FL, USA, January 6-8, 2004
Proceedings

Springer

Volume Editors

Vladimir Lifschitz
University of Texas at Austin
Department of Computer Sciences
1 University Station C0500
Austin, TX 78712, USA
E-mail: vl@cs.utexas.edu

Ilkka Niemelä
Helsinki University of Technology
Dept. of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O. Box 5400, 02015 HUT, Finland
E-mail: Ilkka.Niemela@hut.fi

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): I.2.3, I.2, F.4.1, D.1.6

ISSN 0302-9743
ISBN 3-540-20721-X Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 10975749 06/3142 5 4 3 2 1 0

Preface

The papers in this collection were presented at the *7th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR-7) in Fort Lauderdale, Florida, USA, during January 6–8, 2004. The previous meetings in this series were held in Washington, DC, USA (1991), Lisbon, Portugal (1993), Lexington, USA (1995), Dagstuhl, Germany (1997), El Paso, USA (1999), and Vienna, Austria (2001).

LPNMR conferences are a forum for exchanging ideas on declarative logic programming, nonmonotonic reasoning and knowledge representation. In the 1980s researchers working in the area of nonmonotonic reasoning discovered that their formalisms could be used to describe the behavior of negation as failure in Prolog, and the first LPNMR meeting was convened for the purpose of discussing this relationship. This work has led to the creation of logic programming systems of a new kind, answer set solvers, and to the emergence of a new approach to solving combinatorial search problems, called answer set programming.

The highlights of LPNMR-7 were three invited talks, given by Rina Dechter (University of California, Irvine), Henry Kautz (University of Washington) and Torsten Schaub (University of Potsdam). The program also included 24 regular papers selected after a rigorous review process, 8 system descriptions, and 2 panels.

We would like to thank the Program Committee members and additional reviewers for careful, unbiased evaluation of the submitted papers. We are also grateful to Paolo Ferraris for help with publicizing the Call for Papers, to Fred Hoffman for help with local organizational matters, and to Matti Järvisalo for help with the organization of the electronic Program Committee meeting.

October 2003

Vladimir Lifschitz
Ilkka Niemelä

Conference Organization

Program Co-chairs

Vladimir Lifschitz (University of Texas, USA)
Ilkka Niemelä (Helsinki University of Technology, Finland)

Program Committee

José J. Alferes (New University of Lisbon, Portugal)
Chitta Baral (Arizona State University, USA)
Yannis Dimopoulos (University of Cyprus, Cyprus)
Jürgen Dix (University of Manchester, UK)
Phan Minh Dung (Asian Institute of Technology, Thailand)
Esra Erdem (University of Toronto, Canada)
Michael Gelfond (Texas Tech University, USA)
Tomi Janhunen (Helsinki University of Technology, Finland)
Jorge Lobo (Teltier Technologies, USA)
Fangzhen Lin (Hong Kong University of Science and Technology, Hong Kong)
Ramón P. Otero (University of Corunna, Spain)
Victor W. Marek (University of Kentucky, USA)
Gerald Pfeifer (Vienna University of Technology, Austria)
Enrico Pontelli (New Mexico State University, USA)
Teodor Przymusinski (University of California, Riverside, USA)
Chiaki Sakama (Wakayama University, Japan)
Tran Cao Son (New Mexico State University, USA)
Hudson Turner (University of Minnesota, Duluth, USA)
David S. Warren (University of New York at Stony Brook, USA)
Jia-Huai You (University of Alberta, Canada)

Additional Reviewers

Marcello Balduccini
Stefan Brass
Carlos Damásio
Wolfgang Faber
Katsumi Inoue
Antonis Kakas
Rex Kwok
João Leite
Veena Mellarkod
Bernhard Nebel
Aarati Parmar
David Pearce
Carla Piazza
Inna Pivkina
Axel Polleres
Alessandro Provetti
Halina Przymusinska
Ken Satoh
Mirosław Truszczyński
Dongmo Zhang

Constraints and Probabilistic Networks: A Look At The Interface

Rina Dechter

University of California, Irvine

Abstract. I am going to discuss the interface between two types of networks; one deterministic and one probabilistic. The deterministic network, also known as constraint network, a CSP problem, or a SAT formula, represents a collection of constraints among groups of variables. The probabilistic network is a more organized object, represents a restricted collection of probabilistic relationships among groups of variables. These two paradigms were developed separately in the past 20–30 years and are relatively mature by now, with each paradigm equipped with its own concepts, techniques, heuristics and shortcuts. For example the concept of constraint propagation is unheard of in the probabilistic community. Similarly, notions such as sampling and Monte Carlo simulation (with guaranteed convergence) are rarely examined in constraint processing.
I will start by highlighting conceptual commonalities and differences between the two frameworks, and will propose a simple hybrid framework. I will then talk about benefits that can be obtained by importing techniques from constraint networks to probabilistic networks and back. Finally, if time permits, I will discuss how sampling techniques used in probabilistic networks can inspire algorithms for sampling solution in constraint satisfaction problems.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, p. 1, 2004.

Toward A Universal Inference Engine

Henry Kautz

Department of Computer Science and Engineering
University of Washington

Abstract. In the early days of AI some researchers proposed that intelligent problem solving could be reduced to the application of general purpose theorem provers to an axiomatization of commonsense knowledge. Although automated first-order theorem proving was unwieldy, general reasoning engines for propositional logic turned out to be surprisingly efficient for a wide variety of applications. Still many problems of interest to AI involve probabilities or quantification, and would seem to be beyond propositional methods. However, recent research has shown that the basic backtrack search algorithm for satisfiability generalizes to a strikingly efficient approach for broader classes of inference. We may be on the threshold of achieving the old dream of a universal inference engine.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, p. 2, 2004.

Towards Systematic Benchmarking in Answer Set Programming: The Dagstuhl Initiative

Paul Borchert[1], Christian Anger[1], Torsten Schaub[1⋆], and Mirosław Truszczyński[2]

[1] Institut für Informatik, Universität Potsdam, Postfach 90 03 27, D–14439 Potsdam
{borchi,christian,torsten}@cs.uni-potsdam.de
[2] Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA mirek@cs.uky.edu

1 The Dagstuhl Initiative

Answer-set programming (ASP) emerged in the late 90s as a new logic programming paradigm [3,4,5], having its roots in nonmonotonic reasoning, deductive databases and logic programming with negation as failure. Since its inception, it has been regarded as the computational embodiment of nonmonotonic reasoning and a primary candidate for an effective knowledge representation tool. This view has been boosted by the emergence of highly efficient solvers for ASP [7,2]. It seems now hard to dispute that ASP brought new life to logic programming and nonmonotonic reasoning research and has become a major driving force for these two fields, helping dispell gloomy prophecies of their impending demise.

In September 2002, participants of the Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints, agreed that in order to foster further development of ASP, it is important to establish an infrastructure for benchmarking ASP solvers. The intention was to follow good practices already in place in neighboring fields of satisfiability testing [6] and constraint programming [1]. Thus, the *Dagstuhl Initiative* was born to set up an environment for submitting and archiving benchmarking problems and instances and in which ASP systems can be run under equal and reproducible conditions, leading to independent results.

As the testing ground for different designs of a benchmarking and testing environment for ASP, we used the systems competition at the Dagstuhl Seminar. The following answer set programming systems participated in that initial competition.

- `aspps`, University of Kentucky,
- `assat`, UST Hong Kong,
- `cmodels`, University of Texas,
- `dlv`, Technical University of Vienna,
- `smodels`, Technical University of Helsinki.

⋆ Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 3–7, 2004.

The difficulty that emerged right away was that these systems do not have a common input language nor do they agree on all functionalities. This led to the introduction of three different (major) categories of benchmarks:

Ground: Ground instances of coded benchmarks. As of now, these ground instances are produced by `lparse` or by the `dlv` grounder. These benchmarks can be used to test the performance of ASP solvers accepting as input ground (propositional) programs in output formats of `lparse` or the `dlv` grounder.

Non-Ground: Non-ground programs (that is, with variables) coding benchmark problems. These programs can be used to test the performance of grounders. It is well recognized that significant and by no means negligible effort when solving problems by means of ASP software is spent in grounding.

Free Style: Text descriptions of problems together with concrete (families of) instances given as collections of ground instances. These benchmarks require that users develop their own problem encodings. There are two goal here. First, we want to see how far our solvers can go when faced with hard benchmarks. Second, we hope that best encodings will lead to a set of good programming practices.

While the first two categories rely on the syntax of logic programs, the last category allows for the use of "programming tricks" and system-specific features, like weight and weak constraints. It also supports participation in the effort of systems that are based on other syntax than that of logic programming (for instance, `aspps`).

Within these main categories the following benchmark problems were proposed in Dagstuhl and are now implemented in the present version of the benchmarking system:

- Strategic Companies[1] (Ground, Non-Ground)
- 15-puzzle (Ground, Non-Ground)
- Factoring (Ground)
- Hamiltonian Path (Ground, Non-Ground)
- Schur Numbers (Ground)
- Ramsey Numbers (Non-Ground)
- Same Generation (Non-Ground)
- Coloring (Free Style)
- n-Queens (Free Style)

Clearly, the initial division into categories as well as the set of benchmarks are subject to change and will evolve over time. In fact, one of the features of the system will be to provide to the community a way of submitting new benchmarks and new instances.

[1] Necessitates disjunctive logic programs, or another language of similar expressiveness.

2 The Benchmarking System

The principal goals of the benchmarking system are (1) to provide an infrastructure for accumulating challenging benchmarks, and (2) to facilitate executing ASP solvers under the same conditions, guaranteeing reproducible and reliable performance results.

In the remainder of this section, we sketch the current development state of the benchmarking system and give an outlook on future plans.

2.1 Functionality

The benchmarking system provides the following functionality:

- submitting benchmarking problems, encodings and instances (only registered users)
- installing solvers (only registered users)
- requesting benchmarking runs
- running solvers on benchmarks, independently and in a uniform environment
- testing solvers for correctness against other systems (only registered users)
- displaying results.

2.2 Architecture

We aim at a dynamic system that solves its tasks (running and storing benchmarks) largely without supervision. To achieve this, we have chosen a 2-server architecture, combining an internal server for actually running the benchmarks and an external server for the remaining functionalities including interaction and storage.

The external server is accessible via the Internet. Its main tasks are first to provide database functionalities for handling the benchmark library and second to provide access to the results of running benchmarks in human or machine readable ways. Among others, it is responsible for adding new benchmarking requests, solvers and benchmarking problems. Furthermore, the external server provides user management and a web server. Its central components include a mySQL database storing all information about

Solvers: These are executable answer-set solvers (and auxiliary programs, like parsers and grounders) that may be competing against each other. Stored information includes version, name, path and execution rights.

Call Scripts: These are used to enable suppliers of solvers to ensure their systems are called in the correct way (options, front-ends, etc.). We note that this is the weakest point of the platform since scripts are provided by registered yet external users. Scripts are run with the same privileges a user has. Thus, they need to be hand checked to ensure the unobstructed flow of the benchmarking cycle.

Benchmark Problems: These are text descriptions of benchmark problems and families of corresponding instances (e.g. collections of ground atoms).

Benchmark Encodings: These are logic programs encoding benchmark problems.

Benchmark Instances: These are, usually, ground programs obtained by grounding the union of a program encoding a benchmark problem and the instance description (a set of ground atoms). Non-ground programs are of interest whenever solvers integrating grounding are taken into account.

We note that there is yet no syntax common to all answer-set solvers, even those based on the language of logic programs, and some standardization is necessary.

Benchmark Machine: A description of the system used for benchmarking runs, including data about hardware and software.

Results: Once a solver is run on a benchmark, the results are stored in the database. This part of the database is publicly available via the web interface.

The internal server can only be reached locally. Thus, it is impossible to disturb it from the outside. On this server the actual benchmarking runs take place. It is a largely bare system to minimize side effects (of other software) on the benchmarks. A Perl script is responsible for retrieving benchmark requests from the external servers database and running them. Only one benchmark is run at a time. After a predefined period, the process is killed (if necessary) and completely removed from the system. The next benchmarking request is only handled after a clean environment has been restored. This is very important for obtaining comparable results.

The overall architecture comprising the external and internal server is given in Figure 1.

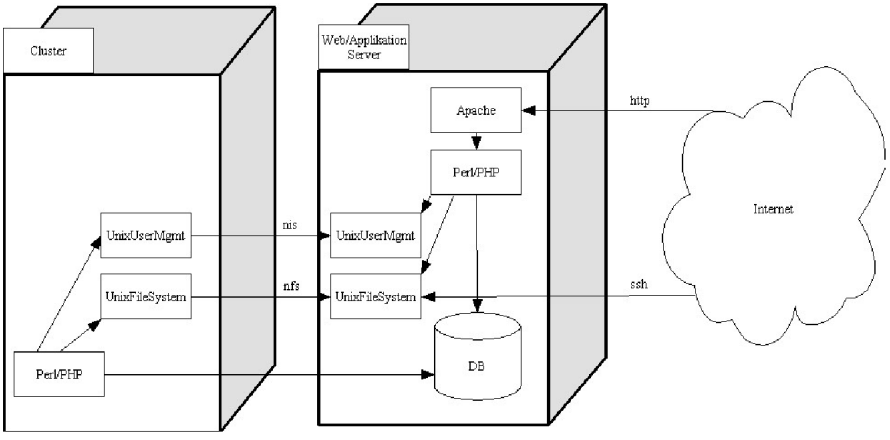


Fig. 1. Two Server Architecture.

2.3 Usage

At this time, the primary user profile is that of a system developer, who wants her system to participate in a system competition. To do so, a developer has to become a registered user, including a system account on the external server. The next step is to upload the solver along with appropriate call scripts, followed by requests for running certain benchmarks. All of this is done via the (upcoming) web interface. A registered user can test his scripts and/or solver on the external server via an ssh connection. Because both servers are kept very similar, this testing on the external server is usually sufficient for guaranteeing executability on the internal server.

Further user profiles, among them benchmark suppliers and independent experimenters, are partially supported and envisaged to be further developed in the future.

Acknowledgments. We would like to thank all participants of the Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints for many stimulating discussions. In particular, we are grateful to the system developers involved in the first system competition for their help, suggestions, and patience with us.

References

1. `csplib. http://4c.ucc.ie/ tw/csplib/.`
2. `dlv. http://www.dbai.tuwien.ac.at/proj/dlv/.`
3. M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing*, 9:365–385, 1991.
4. V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer- Verlag, 1999.
5. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
6. `satlib. http://www.satlib.org/.`
7. `smodels. http://www.tcs.hut.fi/Software/smodels/.`

Semantics for Dynamic Logic Programming: A Principle-Based Approach*

José J. Alferes[1], Federico Banti[1], Antonio Brogi[2], and João A. Leite[1]

[1] CENTRIA, Universidade Nova de Lisboa, Portugal,
{jja,banti,jleite}@di.fct.unl.pt
[2] Dipartimento di Informatica, Università di Pisa, Italy,
brogi@di.unipi.it

Abstract. Over recent years, various semantics have been proposed for dealing with updates in the setting of logic programs. The availability of different semantics naturally raises the question of which are most adequate to model updates. A systematic approach to face this question is to identify general principles against which such semantics could be evaluated. In this paper we motivate and introduce a new such principle – the *refined extension principle* – which is complied with by the stable model semantics for (single) logic programs. It turns out that none of the existing semantics for logic program updates, even though based on stable models, complies with this principle. For this reason, we define a refinement of the dynamic stable model semantics for Dynamic Logic Programs that complies with the principle.

1 Introduction and Motivation

Most of the research in the field of logic programming for representing knowledge that evolves with time has focused on changes in the extensional part of knowledge bases (factual events or observations). This is what happens with the event calculus [10], logic programming forms of the situation calculus [15,17] and logic programming representations of action languages [9]. In all of these, the problem of updating the intensional part of the knowledge base (rules or action descriptions) remains basically unexplored.

In recent years, some amount of effort was devoted to explore the problem of updates in a logic programming setting leading to different framework proposals and semantics [1,4,6,11,13,14,19,21]. According to these proposals, knowledge is given by a sequence of logic programs (or a Dynamic Logic Program) where each is to be viewed as an update to the previous ones. Most of the existing semantics are based on the notion of causal rejection of rules [13] i.e., the rejection of a prior rule if there is a newer one that conflicts with it, and on a notion of default assumptions that can be added to the theory. Different notions of rejection

* This work was partially supported by FEDER financed project FLUX (POSI/40958/SRI/2001) and by project SOCS (IST-2001-32530). Special thanks are due to Pascal Hitzler and Reinhard Kahle for helpful discussions.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 8–20, 2004.

and default assumptions lead to different semantics, namely the justified update semantics [13,14], the dynamic stable model semantics [1,2,11] and the update answer-set semantics [6][1]. While such existing semantics based on causal rejection coincide on a large class of program updates, they essentially differ on the extent to which they are immune to some apparently inoffensive updates (e.g. tautologies[2]). Take, for example, the program $P_1 = \{a.\}$ and update it with $P_2 = \{not\, a \leftarrow not\, a\}$. Intuitively one would expect the update of P_1 with P_2 not to change the semantics because the only rule of P_2 is a tautology. This is not the case according to the semantics of justified updates which admits, after the update, the models $\{a\}$ and $\{\}$. Similar behaviours are exhibited by the update answer-set semantics [6]. Examples such as this one were one of the main reasons for the introduction of the dynamic stable model semantics [1,2] which properly deals with them. Unfortunately, there still remain examples involving tautological updates where none of the existing semantics behaves as expected. Let us now show an example to illustrate the problem.

Example 1. Consider the program P_1 describing some knowledge about the sky. At each moment it is either day time or night time, we can see the stars whenever it is night time and there are no clouds, and currently it is not possible to see the stars.

$$\begin{array}{ll} P_1 : day \leftarrow not\, night. & stars \leftarrow night, not\, cloudy. \\ \quad night \leftarrow not\, day. & \quad not\, stars. \end{array}$$

The only dynamic stable model of this program is $\{day\}$. Suppose now the program is updated with the following tautology:

$$P_2 : stars \leftarrow stars.$$

This tautological update introduces the new dynamic stable model $\{night, stars\}$. Furthermore these results are shared by all other semantics for updates based on causal rejection [1,4,6,11,13,14]. We argue that this behaviour is counterintuitive as the addition of the tautology in P_2 should not add new models.

This alone should be enough to have us start a quest for a semantics for updates that is immune to tautologies. But the problem runs deeper. Typically, these tautological updates are just particular instances of more general updates that should be ineffective but, in reality, cause the introduction of new models e.g. those with a rule whose head is *self-dependent*[3] as in the following example:

Example 2. Consider again program P_1 of Example 1, and replace P_2 with

$$P_2 : stars \leftarrow venus. \qquad venus \leftarrow stars.$$

While P_1 has only one model (viz., $\{day\}$), according to all the existing semantics for updates based on causal rejection, the update P_2 adds a second model,

[1] In this paper, we only consider semantics based on the notion of causal rejection.
[2] By a tautology we mean a rule of the form $L \leftarrow Body$ with $L \in Body$.
[3] For the definition of self-dependent literals in a logic program, see [3].

$\{night, stars, venus\}$. Intuitively speaking, this new model arises since the update P_2 causally rejects the rule of P_1 which stated that it was not possible to see the stars.

On the basis of these considerations, it is our stance that, besides the principles used to analyze and compare these semantics, described in [6,11], another important principle is needed to test the adequacy of semantics of logic program updates in some important situations, in particular those concerning the unwanted generation of new dynamic stable models when certain sets of rules are added to a dynamic logic program. It is worth noting that an update with the form of P_2 in Example 2 may have the effect of eliminating previously existing models, this often being a desired effect, as illustrated by the following example:

Example 3. Consider program P_1 with the obvious intuitive reading: one is either alone or with friends, and one is either happy or depressed.

$$P_1 : friends \leftarrow not\, alone. \qquad happy \leftarrow not\, depressed.$$
$$alone \leftarrow not\, friends. \qquad depressed \leftarrow not\, happy.$$

This program has four dynamic stable models namely, $\{friends, depressed\}$, $\{friends, happy\}$, $\{alone, happy\}$ and $\{alone, depressed\}$. Suppose now the program is updated with the following program (similar to P_2 used in Example 2):

$$P_2 : depressed \leftarrow alone. \qquad alone \leftarrow depressed.$$

This update specified by P_2 eliminates two of the dynamic stable models, leaving only $\{friends, happy\}$ and $\{alone, depressed\}$, this being a desirable effect.

In this paper we propose a new principle, that we call the *refined extension principle*, which can be used to compare different semantics for updates based on the stable model semantics — as is the case of all the above mentioned.

To this purpose, we start with the simple case of a single logic program and set forth the *refined extension principle* which, if complied with by a semantics, specifies some conditions under which rules can be safely added without introducing new models according to that semantics. Notably, the stable model semantics [8] complies with this principle. Informally, the semantics based on stable models can be obtained by taking the least Herbrand model of the definite program obtained by adding some assumptions (default negations) to the initial program. Intuitively speaking, the refined extension principle states that the addition of rules that do not change that least model should not lead to obtaining more (stable) models.

Subsequently, we generalize this principle by lifting it to the case of semantics for dynamic logic programs. Not unexpectedly, given the examples above, it turns out that none of the existing semantics for updates based on causal rejection complies with such principle, which leads us to introduce a new semantics for dynamic logic programs, namely the *refined dynamic stable model semantics*, which complies with the refined extension principle. The *refined dynamic stable model semantics* is obtained by refining the dynamic stable model semantics [1,2,

11] which, of all the existing semantics, as shall be seen, is the one that complies with the refined extension principle on the largest class of programs.

The rest of the paper is organized as follows. Section 2 recalls some preliminary notions and establishes notation. Section 3 is devoted to motivate and present the refined extension principle, while in Section 4 a refined semantics for logic program updates that complies with the principle is presented. Section 5 is devoted to compare the new semantics with other existing semantics, and to analyze these with respect to the refined extension principle. Finally, some concluding remarks are drawn.

2 Preliminaries

In this paper we extensively use the concept of generalized logic programs [16] i.e. logic programs that allow for default negation both in the bodies as well as in the heads of their rules. A generalization of the stable models semantics for normal logic programs [8] to the class of generalized programs was defined by Lifschitz and Woo [16]. Here we present such semantics differently from [16], the equivalence of both definition being proven in [2].

Let $\mathcal{A}$ be a set of propositional **atoms**. A **default literal** is an atom preceded by *not*. A **literal** is either an atom or a default literal. A **rule** r is an ordered pair $H(r) \leftarrow B(r)$ where $H(r)$ (dubbed the head of the rule) is a literal and $B(r)$ (dubbed the body of the rule) is a finite set of literals. A rule with $H(r) = L_0$ and $B(r) = \{L_1, \ldots, L_n\}$ will simply be written as $L_0 \leftarrow L_1, \ldots, L_n$. A rule with $H(r) = L_0$ and $B(r) = \{\}$ is called a fact and will simply be written as L_0. A **generalized logic program** (*GLP*) P, in $\mathcal{A}$, is a finite or infinite set of rules. By $P_\emptyset$ we mean an empty set of rules. If $H(r) = A$ (resp. $H(r) = not\, A$) then $not\, H(r) = not\, A$ (resp. $not\, H(r) = A$). Two rules r and r' are **conflicting**, denoted by $r \bowtie r'$, iff $H(r) = not\, H(r')$.

An **interpretation** M of $\mathcal{A}$ is a set of atoms ($M \subseteq \mathcal{A}$). An atom A is true in M, denoted by $M \vDash A$, iff $A \in M$, and false otherwise. A default literal $not\, A$ is true in M, denoted by $M \vDash not\, A$, iff $A \notin M$, and false otherwise. A set of literals B is true in M, denoted by $M \vDash B$, iff each literal in B is true in M.

An interpretation M of $\mathcal{A}$ is a **stable model** (or answer set) of a generalized logic program P iff

$$M' = least\,(P \cup \{not\, A \mid A \notin M\})$$

where $M' = M \cup \{not\_A \mid A \notin M\}$, A is an atom, and $least(.)$ denotes the least model of the definite program obtained from the argument program by replacing every default literal $not\, A$ by a new atom not_A[4].

A **dynamic logic program** (*DLP*) is a sequence of generalized logic programs. Let $\mathcal{P} = (P_1, ..., P_s)$ and $\mathcal{P}' = (P'_1, ..., P'_s)$ be two DLPs. We use $\rho(\mathcal{P})$ to denote the multiset of all rules appearing in the programs $P_1, ..., P_s$, and $\mathcal{P} \cup \mathcal{P}'$

[4] This amounts to determining the least model of the argument program, treating default literals as positive atoms.

to denote the DLP $(P_1 \cup P'_1, ..., P_s \cup P'_s)$. According to all semantics based on causal rejection of rules, an interpretation M models a DLP iff

$$M' = \Gamma(\mathcal{P}, M)$$

where

$$\Gamma(\mathcal{P}, M) = least(\rho(\mathcal{P}) - Rej(\mathcal{P}, M) \cup Def(\mathcal{P}, M))$$

and where $Rej(\mathcal{P}, M)$ stands for the set of rejected rules and $Def(\mathcal{P}, M)$ for the set of default assumptions, both given $\mathcal{P}$ and M. Intuitively, we first determine the set of rules from $\mathcal{P}$ that are not rejected, i.e. $\rho(\mathcal{P}) - Rej(\mathcal{P}, M)$, to which we add a set of default assumptions $Def(\mathcal{P}, M)$. Note the similarity to the way stable models of generalized logic programs are obtained, where default assumptions of the form $not\ A$ are added for every $A \notin M$.

From [11] it is easy to see that all existing semantics for updates based on causal rejection are parameterizable using different definitions of $Rej(\mathcal{P}, M)$ and $Def(\mathcal{P}, M)$. The **dynamic stable model** semantics for DLP [1,11] is obtained with the following definitions:

$$Rej(\mathcal{P}, M) = \{r \mid r \in P_i, \exists r' \in P_j, i < j, r \bowtie r', M \vDash B(r')\}$$
$$Def(\mathcal{P}, M) = \{not\ A \mid \nexists r \in \rho(\mathcal{P}), H(r) = A, M \vDash B(r)\}$$

3 Refined Extensions

As mentioned in the Introduction, we are interested in exploring conditions guaranteeing that the addition of a set of rules to a (dynamic) logic program does not generate new (dynamic) stable models. In this Section, we motivate and introduce the notion of refined extension for both the case of single logic programs and dynamic logic programs, which, together with the results proven, constitute a step in such direction.

3.1 Refined Extensions of Generalized Logic Programs

Informally, the semantics based on stable models are obtained by taking the least Herbrand model of the definite program obtained by adding some assumptions (default negations) to the initial program i.e., the stable models of a generalized logic program P are those interpretations M such that M' coincides with the least Herbrand model of the program[5] $P \cup \{not\ A \mid A \notin M\}$. In general, several semantics share the characteristic that the models of a program P can be characterized as the least Herbrand model of the program $P \cup Assumptions(P, M)$, where $Assumptions(P, M)$ is simply a set of default literals whose definition depends on the semantics in use. Note that all of the stable models [8], the well-founded [7] and the weakly perfect model semantics [18] can be defined in this

[5] Here and elsewhere, whenever we mention the Herbrand model of a GLP, we mean the Herbrand model of the definite logic obtained from the GLP by replacing every default literal $not\ A$ by a new atom not_A.

way. As mentioned above, the stable model semantics of normal and generalized programs can be obtained by establishing that

$$Assumptions\,(P, M) = \{not\ A \mid A \notin M\}\,.$$

In the sequel, if Sem is a semantics definable in such a way, by $Sem(P)$ we denote the set of all models of a given program P, according to Sem.

With the intention of defining the refined extension principle, we first need to set forth some intermediate notions, namely that of *syntactic extension* of a program. Intuitively we say that $P \cup E$ is a syntactic extension of P iff the rules in E have no effect on the least Herbrand model of P. Formally:

Definition 1. *Let P and E be generalized logic programs. We say that $P \cup E$ is a* ***syntactic extension*** *of P iff $least\,(P) = least\,(P \cup E)$.*

Consider now a generalized program P, and a set of rules E. A model M of $P \cup E$ in the semantic Sem is computed as the least Herbrand model of the definite logic program obtained by adding the set of default assumptions to $P \cup E$. We can then apply the concept of syntactical extension to verify whether the addition of the rules in E does not influence the computation of M. If this is the case for all models of the program $P \cup E$, according to Sem, we say that $P \cup E$ is a *refined extension* of the original program P. Formally:

Definition 2. *Let P and E be generalized logic program, M an interpretation, and Sem a semantics for generalized logic programs. We say that $P \cup E$ is an* ***extension*** *of P with respect to Sem and M iff*

$$P \cup Assumptions\,(P \cup E, M) \cup E$$

is a syntactic extension of $P \cup Assumptions(P \cup E, M)$. We say that $P \cup E$ is a ***refined extension*** *of P with respect to Sem iff $P \cup E$ is an extension of P with respect to Sem and all models in $Sem(P \cup E)$.*

Example 4. Let P_1 and P_2 be the programs:

$$\begin{array}{lll} P_1 : day \leftarrow not\ night. & & stars \leftarrow night, not\ cloudy. \\ \quad night \leftarrow not\ day. & & not\ stars. \\ P_2 : stars \leftarrow stars & & \end{array}$$

It is clear that no matter what the added assumptions ($Assumptions\,(P_1 \cup P_2, M)$) are, given any model M, the least model of $P_1 \cup Assumptions\,(P_1 \cup P_2, M)$ is the same as the least model of $P_1 \cup Assumptions\,(P_1 \cup P_2, M) \cup P_2$. Thus, $P_1 \cup P_2$ is a refined extension of P_1.

We are now ready to formulate the refined extension principle for semantics of generalized logic programs. Intuitively, a semantics complies with the refined extension principle iff a refined extension of a program P does not have more models than P.

Principle 1 (Refined extension – static case) *A semantics Sem for generalized logic programs complies with the refined extension principle iff for any two generalized logic programs P and E, if $P \cup E$ is a refined extension of P then*

$$Sem(P \cup E) \subseteq Sem(P).$$

As one may expect, the principle properly deals with the case of adding tautologies, i.e., for any semantics *Sem* that complies with the principle, the addition of tautologies does not generate new models.

Proposition 1. *Let Sem be a semantics for generalized programs, P a generalized program, and τ a tautology. If Sem complies with the refined extension principle then*

$$Sem(P \cup \{\tau\}) \subseteq Sem(P).$$

Most importantly, the stable model semantics complies with the refined extension principle, as stated in the following proposition:

Proposition 2. *The stable model semantics complies with the refined extension principle.*

As an immediate consequence of these two propositions, we get that the addition of tautologies to a generalized program does not introduce new stable models. The converse is also true i.e., the addition of tautologies to a generalized program does not eliminate existing stable models.

3.2 Refined Extensions of Dynamic Logic Programs

We now generalize the refined extension principle to the case of dynamic logic programs, so as to guarantee that certain updates do not generate new models.

Definition 3. *Let $\mathcal{P}$ and $\mathcal{E}$ be two dynamic logic programs*[6]*, Sem a semantics for dynamic logic programs and M an interpretation. We say that $\mathcal{P} \cup \mathcal{E}$ is an* ***extension*** *of $\mathcal{P}$ with respect to M iff*

$$[\rho(\mathcal{P}) - Rej(\mathcal{P} \cup \mathcal{E}, M)] \cup Def(\mathcal{P} \cup \mathcal{E}, M) \cup [\rho(\mathcal{E}) - Rej(\mathcal{P} \cup \mathcal{E}, M)]$$

is a syntactical extension of

$$[\rho(\mathcal{P}) - Rej(\mathcal{P} \cup \mathcal{E}, M)] \cup Def(\mathcal{P} \cup \mathcal{E}, M)$$

We say that $\mathcal{P} \cup \mathcal{E}$ is a ***refined extension*** *of $\mathcal{P}$ iff $\mathcal{P} \cup \mathcal{E}$ is an extension of $\mathcal{P}$ with respect to all models in $Sem(\mathcal{P} \cup \mathcal{E})$.*

Note that the above definition is the straightforward lifting of Definition 2 to the case of dynamic logic programs. Roughly speaking, we simply replaced P and E with $\rho(\mathcal{P}) - Rej(\mathcal{P} \cup \mathcal{E}, M)$ and $\rho(\mathcal{E}) - Rej(\mathcal{P} \cup \mathcal{E}, M)$, respectively.

The refined extension principle is then formulated as follows.

[6] Here, and elsewhere, we assume that the sequences of GLPs (or DLPs) $\mathcal{P}$ and $\mathcal{E}$ are of the same length.

Principle 2 (Refined extension principle) *A semantics Sem for dynamic logic programs complies with the refined extension principle iff for any dynamic logic programs $\mathcal{P}$ and $\mathcal{P} \cup \mathcal{E}$, if $\mathcal{P} \cup \mathcal{E}$ is a refined extension of $\mathcal{P}$ then*

$$Sem(\mathcal{P} \cup \mathcal{E}) \subseteq Sem(\mathcal{P}).$$

Unfortunately, as we already pointed out in the Introduction, none of the existing semantics for dynamic logic programs based on causal rejection comply with the refined extension principle.

Example 5. Consider again the program P_1 and P_2 of Example 2. The presence of the new model contrasts with the refined extension principle. Indeed, if we consider the empty update $P_\emptyset$, then the dynamic logic program $(P_1, P_\emptyset)$ has only one stable model (viz., $\{day\}$). Since, as the reader can check, (P_1, P_2) is a refined extension of $(P_1, P_\emptyset)$ then, according to the principle, all models of (P_1, P_2) should also be models of $(P_1, P_\emptyset)$. This is not the case for existing semantics.

As for the case of generalized programs, if we consider a semantics *Sem* for dynamic logic programs that complies with the principle, the addition of tautologies does not generate new models. This is stated in the following proposition that lifts Proposition 1 to the case of dynamic logic programs.

Proposition 3. *Let Sem be a semantics for dynamic logic programs, $\mathcal{P}$ a dynamic logic program, and $\mathcal{E}$ a sequence of sets of tautologies. If Sem complies with the refined extension principle then*

$$Sem(\mathcal{P} \cup \mathcal{E}) \subseteq Sem(\mathcal{P}).$$

4 Refined Semantics for Dynamic Logic Programs

In this Section we define a new semantics for dynamic logic programs that complies with the refined extension principle. Before proceeding we will take a moment to analyze the reason why the dynamic stable model semantics fails to comply with the refined extension principle in Example 1. In this example, the extra (counterintuitive) stable model $\{night, stars\}$ is obtained because the tautology $stars \leftarrow stars$ in P_2 has a true body in that model, hence rejecting the fact $not\, stars$ of P_1. After rejecting this fact, it is possible to consistently conclude $stars$, and thus verify the fixpoint condition, via the rule $stars \leftarrow night, not\, cloudy$ of P_1.

Here lies the matrix of the undesired behaviour exhibited by the dynamic stable model semantics: One of the two conflicting rules in the same program (P_1) is used to support a later rule (of P_2) that actually removes that same conflict by rejecting the other conflicting rule. Informally, rules that should be irrelevant may become relevant because they can be used by one of the conflicting rules to defeat the other.

A simple way to inhibit this behaviour is to let conflicting rules in the same state inhibit each other. This can be obtained with a slight modification to

the notion of rejected rules of the dynamic stable model semantics, namely by also allowing rules to reject other rules in the same state. Since, according to the dynamic stable model semantics, rejected rules can reject other rules, two rules in the same state can reject each other, thus avoiding the above described behaviour.

Definition 4. *Let $\mathcal{P}$ be a dynamic logic program and M an interpretation. M is a* refined dynamic stable model *of $\mathcal{P}$ iff*

$$M' = \Gamma^S(\mathcal{P}, M)$$

where

$$\Gamma^S(\mathcal{P}, M) = least\left(\rho(\mathcal{P}) - Rej^S(\mathcal{P}, M) \cup Def(\mathcal{P}, M)\right)$$

and

$$Rej^S(\mathcal{P}, M) = \{r \mid r \in P_i, \exists r' \in P_j, i \leq j, r \bowtie r', M \vDash B(r')\}$$

At first sight this modification could seem to allow the existence of models in cases where a contradiction is expected (e.g. in a sequence where the last program contains facts for both A and $not\ A$): if rules in the same state can reject each other then the contradiction is removed, and the program could have undesirable models. Notably, the opposite is actually true (cf. theorem 4 below), and the refined dynamic stable models are always dynamic stable models, i.e., allowing the rejection of rules by rules in the same state does not introduce extra models. Consider a DLP $\mathcal{P}$ with two conflicting rules (with heads A and $not\ A$) in one of its programs P_i. Take an interpretation M where the bodies of those two rules are both true (as nothing special happens if a rule with false body is rejected) and check if M is a refined dynamic stable model. By definition 4, these two rules reject each other, and reject all other rules with head A or $not\ A$ in that or in any previous state. Moreover, $not\ A$ cannot be considered as a default assumption, i.e., does not belong to $Def(\mathcal{P}, M)$. This means that all the information about A with origin in P_i or any previous state is deleted. Since M' must contain either A or not_A, the only possibility for M to be a stable model is that there exists a rule τ in some later update whose head is either A or $not\ A$, and whose body is true in M. This means that a potential inconsistency can only be removed by some later update.

Finally, as this was the very motivation for introducing the refined semantics, it is worth observing that the refined semantics does comply with the refined extension principle, as stated by the following theorem.

Theorem 3. *The refined dynamic stable model semantics complies with the refined extension principle.*

By Proposition 3, it immediately follows from this theorem that the addition of tautologies never adds models in this semantics. Note that the converse is also true: the addition of tautologies does not eliminate existing models in the refined semantics i.e., the refined dynamic stable model semantics is immune to tautologies. Moreover the refined semantics preserves all the desirable properties of the previous semantics for dynamic logic programs [6,11].

To give an insight view of the behaviour of the refined semantics, we now illustrate how the counterintuitive results of example 1 are eliminated.

Example 6. Consider again the DLP $\mathcal{P} = (P_1, P_2)$ of example 1

$$\begin{array}{ll} P_1 : day \leftarrow not\, night. & stars \leftarrow night, not\, cloudy. \\ \quad night \leftarrow not\, day. & not\, stars. \\ P_2 : stars \leftarrow stars & \end{array}$$

This DLP has one refined dynamic stable model, $M = \{day\}$. Thus the conclusions of the semantics match with the intuition that it is day and it is not possible to see the stars. We now show that M is a refined dynamic stable model. First of all we compute the sets $Rej^S(\mathcal{P}, M)$ and $Def(\mathcal{P}, M)$:

$$\begin{aligned} Rej^S(\mathcal{P}, M) &= \{stars \leftarrow night, not\, cloudy.\} \\ Def(\mathcal{P}, M) &= \{not\, night, not\, stars, not\, cloudy\} \end{aligned}$$

Then we check whether M is a refined dynamic stable model according to definition 4. Indeed:

$$\begin{aligned} \Gamma^S(\mathcal{P}, M) &= least\left((P_1 \cup P_2) - Rej^S(\mathcal{P}, M) \cup Def(\mathcal{P}, M)\right) = \\ &= \{day, not\_night, not\_stars, not\_cloudy\} = M' \end{aligned}$$

As mentioned before, the dynamic stable model semantics, besides M, also admits the interpretation $N = \{night, stars\}$ as one of its models, thus violating the refined extension principle. We now show that N is not a refined dynamic stable model. As above we compute the sets:

$$\begin{aligned} Rej^S(\mathcal{P}, N) &= \{not\, stars; \quad stars \leftarrow night, not\, cloudy.\} \\ Def(\mathcal{P}, N) &= \{not\, day, not\, cloudy\} \end{aligned}$$

Hence:

$$\begin{aligned} \Gamma^S(\mathcal{P}, N) &= least\left((P_1 \cup P_2) - Rej^S(\mathcal{P}, N) \cup Def(\mathcal{P}, N)\right) = \\ &= \{night, not\_day, not\_cloudy\} \neq N' \end{aligned}$$

From where we conclude, according to definition 4, that N is not a refined dynamic stable model.

5 Comparisons

Unlike the refined dynamic stable model semantics presented here, none of the other existing semantics based on causal rejection respect the refined extension principle and, consequently, none is immune to the addition of tautologies.

It is clear from the definitions that the refined semantics coincides with the dynamic stable model semantics [1,2,11] for sequences of programs with no conflicting rules in a same program. This means that the dynamic stable model semantics complies with the refined extension principle for such class of sequences of programs, and would have no problems if one restricts its application to that class. However, such limitation would reduce the freedom of the programmer, particulary in the possibility of using conflicting rules to represent integrity

constraints. Another limitation would result from the fact that updates also provide a tool to remove inconsistency in programs by rejecting conflicting rules. Such feature would be completely lost in that case.

With respect to the other semantics based on causal rejection, it is not even the case that the principles is satisfied by sequences in that restricted class. The update answer-set semantics [6] (as well as inheritance programs of [4]), and the justified update semantics [13,14] fail to be immune to tautologies even when no conflicting rules occur in the same program:

Example 7. Consider the DLP $\mathcal{P} = (P_1, P_2, P_3)$ where (taken from [11]):

$$P_1 : day. \qquad P_2 : not\ day. \qquad P_3 : day \leftarrow day.$$

stating that initially it is day time, then it is no longer day time, and finally (tautologically) stating that whenever it is day time, it is day time. While the semantics of justified updates [13,14] and the dynamic stable model semantics [1,11] select $\{day\}$ as the only model, the update answer set semantics of [6] (as well as inheritance programs of [4]) associates two models, $\{day\}$ and $\{\}$, with such sequence of programs[7].

While the semantics of justified updates [13,14] works properly for the above example, there are classes of program updates for which it does not comply with the refined extension principle:

Example 8. Consider the DLP $\mathcal{P} = (P_1, P_2)$ where (taken from [11]):

$$P_1 : day. \qquad P_2 : not\ day \leftarrow not\ day.$$

According to the semantics of justified updates, (P_1, P_2) has two models, $M_1 = \{day\}$ and $M_2 = \{\}$, whereas $(P_1, P_\emptyset)$ has the single model M_1, thus violating the refined extension principle.

Finally, observe that the refined semantics is more credulous than all the other semantics, in the sense that the set of its models is always a subset of the set of models obtained with any of the others thus making its intersection larger. Comparing first with the dynamic stable model semantics:

Theorem 4. *Let $\mathcal{P}$ be a DLP, and M an interpretation. If M is a refined dynamic stable model then M is a dynamic stable model.*

This result generalizes to all the other semantics since the dynamic stable model is the most credulous of the existing semantics. Indeed, each dynamic stable model is also a model in the justified update semantics [11]. Inheritance programs are defined for disjunctive logic programs, but if we restrict to the non disjunctive case, this semantics coincides with the update answer-set semantics of [6], and each dynamic stable model is also an update answer-set [11].

[7] Notice that, strictly speaking, the semantics [4,6] are actually defined for extended logic programs, with explicit negation, rather than for generalized programs. However, the example can be easily adapted to extended logic programs.

The analysis of the semantics for updates that are not based on causal rejection (e.g. [19,21]) is beyond the scope of this paper. Even though the refined extension principle is not directly applicable to evaluate such semantics, they do not appear satisfactory in the way they deal with simple examples, as shown in [6,11] where a deeper analysis of such semantics can be found.

6 Concluding Remarks

We have started by motivating and introducing a new general principle – the refined extension principle – that can be used to compare semantics of logic programs that are obtainable from the least model of the original program after the addition of some (default) assumptions. For normal logic programs, both the well-founded and stable models are obtainable as such. The principles states that the addition of rules that do not change the least model of the program plus assumptions can never generate new models. A special case of this principle concerns the addition of tautologies. Not surprisingly, the stable model semantics for normal and generalized logic programs complies with this principles.

We have generalized this principle for the case of dynamic logic programs, noticing that none of the existing semantics complies with it. A clear sign of this, already mentioned in the literature [6,11], was the fact that none of these existing semantics is immune to tautologies. We have shown that, among these existing semantics, the dynamic stable model semantics [1,2,11] is the one that complies with the principle for a wider class, viz., the class of dynamic logic programs without conflicting rules in a same program in the sequence.

To remedy this problem exhibited by the existing semantics, and guided by the refined extension principle, we have introduced a new semantics for dynamic logic programs – the refined dynamic stable model semantics – and shown that it complies with such principle. Furthermore, we have obtained that this semantics is immune to tautologies. We have compared the new semantics with extant ones and have shown that, besides the difference regarding the principle, this semantics is more credulous than any of the others, in the sense of admitting a smaller set of stable models (thus yielding a larger intersection of stable models).

Future lines of research include extending this study to deal with multi-dimensional updates [11,12]. Multi-dimensional updates can be viewed as a tool for naturally encoding and combining knowledge bases that incorporate information from different sources, which may evolve in time. Existing semantics suffer from the same kind of problems we identified here for dynamic logic program.

Another important line for future work is the implementation of the refined semantics. We intend to do it by finding a transformation from dynamic programs to generalized programs, such that the stable models of the latter are in a one-to-one correspondence with the refined dynamic stable models of the former, similarly to what has been done for e.g., the dynamic stable semantics. The existence of such a transformation would directly provide a means to implement the refined dynamic stable model semantics of dynamic logic programs, by

resorting to an implementation for answer-set programming, such as SMODELS [20] or DLV [5].

References

1. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic logic programming. In *Procs. of KR'98*. Morgan Kaufmann, 1998.
2. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, September/October 2000.
3. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19 & 20:9–72, May 1994.
4. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In *Procs. of ICLP'99*. MIT Press, 1999.
5. DLV. The DLV project - a disjunctive datalog system (and more), 2000. Available at `http://www.dbai.tuwien.ac.at/proj/dlv/`.
6. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2(6), 2002.
7. 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.
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *Procs. of ICLP'88*. MIT Press, 1988.
9. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.
10. R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
11. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
12. J. A. Leite, J. J. Alferes, and L. M. Pereira. Multi-dimensional dynamic knowledge representation. In *Procs. of LPNMR'01*, volume 2173 of *LNAI*. Springer, 2001.
13. J. A. Leite and L. M. Pereira. Generalizing updates: From models to programs. In *Procs. of LPKR'97*, volume 1471 of *LNAI*. Springer Verlag, 1997.
14. J. A. Leite and L. M. Pereira. Iterated logic program updates. In *Procs. of JICSLP'98*. MIT Press, 1998.
15. H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 03, 1998.
16. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In *Procs of KR'92*. Morgan-Kaufmann, 1992.
17. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.
18. H. Przymusinska and T. Przymusinski. Weakly perfect model semantics. In *Procs. of ICLP'88*. MIT Press, 1988.
19. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In *Procs. of LPNMR'99*, volume 1730 of *LNAI*. Springer.
20. SMODELS. The SMODELS system, 2000. Available at `http://www.tcs.hut.fi/Software/smodels/`.
21. Y. Zhang and N.Y. Foo. Updating logic programs. In *Procs of ECAI'98*. John Wiley & Sons, 1998.

Probabilistic Reasoning With Answer Sets

Chitta Baral[1], Michael Gelfond[2⋆], and Nelson Rushton[2⋆]

[1] Department of Computer Science and Engineering
Arizona State University
Tempe, Arizona 85287
chitta@asu.edu
[2] Department of Computer Science
Texas Tech University
Lubbock, Texas 79409
mgelfond@cs.ttu.edu
nrushton@coe.ttu.edu

Abstract. We give a logic programming based account of probability and describe a declarative language P-log capable of reasoning which combines both logical and probabilistic arguments. Several non-trivial examples illustrate the use of P-log for knowledge representation.

1 Introduction

A man is sitting at a blackjack table, where cards are being dealt from a single deck. What is the probability he is dealt a blackjack (two cards, one of which is an ace, and the other of which is a 10 or a face card)? The standard answer is $4 * 16/C(52, 2)$. Now suppose that on the previous hand, cards removed from the deck were a king, two 3's, an 8 and a 5. This changes the resulting calculation – but only for someone who saw the cards dealt, and takes them into account. Considering more information could change the result even further. In fact, the probability the player receives a blackjack will be either 1 or 0 if we take into account the arrangement of the already-shuffled cards lying in the shoe.

This simple example illustrates an important point: In order to be well posed, questions about probabilities must be asked and answered with respect to a body of knowledge. In this paper we introduce P-log, a language for representing such knowledge. P-log allows the user to represent both logical knowledge and basic probabilistic information about a domain; and its semantics provides a mechanism for systematically deriving conditional and unconditional probabilities from the knowledge represented. P-log uses A-Prolog[1] or its dialects to express logical knowledge. Basic probabilistic information is expressed by probability atoms, say $pr(a|_c\, B) = v$, which is read intuitively as saying *a is caused by factors determined by* B *with probability* v. As noted in [15], causal probabilities

⋆ We want to thank the reviewers for useful comments. The last two authors were partially supported by NASA under grants NCC9-157, NAG2-1560.

[1] The language of logic programs with classical and default negation and disjunction under the answer set semantics [4].

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 21–33, 2004.

differ from ordinary conditional probabilities in two respects. First, a causal probability statement implicitly represents a set of conditional independence assumptions: given its cause C, an effect E is probabilistically independent of all factors except the (direct or indirect) effects of E. Second, causal probabilities can be used to determine the effects of actions which interrupt the normal mechanisms of a model, while conditional probabilities cannot do this in general (see Example 4). Both of these differences are captured in the semantics of P-log.

2 The P-log Language

2.1 Syntax of P-log

Let $\mathcal{L}$ be a dialect of A-Prolog (e.g. [12,13,3,2]). A *probabilistic logic program* (P-log program), Π, over $\mathcal{L}$ consists of *sorted signature*, *declarations*, *regular rules* of $\mathcal{L}$, *probabilistic information*, *observations*, and *actions*.

Signature: The sorted signature Σ of Π contains sets $O, F,$ and R of object, function, and relation names respectively. We assume F is the disjoint union of sets F_r and F_a. Members of F_a will be called *attributes*. Terms will be formed as usual from O and F_r, and atoms as usual from R and the set of terms. In addition, we allow atoms of the form $a(\bar{t}) = t_0$, where t_0 is a term, $\bar{t}$ a vector of terms, and $a \in F_a$. Terms and literals are normally denoted by (possibly indexed) letters t and l respectively; $\bar{t}$ stands for a vector of terms. Letters c's, a's, and r's will be used as generic names for sorts, attributes and relations respectively. Other lower case letters will denote objects; capital letters will stand for variables. A rule with variables will be viewed as a shorthand for the collection of its ground instances (with variables replaced by properly sorted ground terms).

The **declaration** of a P-log program is a collection of definitions of sorts, and typing information for attributes and relations.

A sort c can be defined by explicitly listing its elements, $c = \{x_1, \ldots, x_n\}$, or by a logic program with a unique answer set A. In the latter case $x \in c$ iff $c(x) \in A$. A statement

$$\text{rel } r : c_1 \times \cdots \times c_n \tag{1}$$

specifies sorts for parameters of n-ary relation r. The domain and range of an attribute a are given by a statement

$$a : c_1 \times \cdots \times c_n \rightarrow c_0 \tag{2}$$

If $n = 0$ we simply write $rel\ r$ and $a : c_0$ respectively.

The following example will be used throughout this section.

Example 1. Consider a domain containing two dice. A P-log program Π_0 modeling the domain will have a signature Σ containing the names of the two dice, d_1 and d_2, an attribute $roll$ mapping each die into its value, an integer from 1 to 6, relations $owns(D, P)$, $even(D)$, and $even$ where P and D range over the sorts $person$ and $dice$ respectively,

and "imported" arithmetic functions $+$ and mod. The corresponding declarations, D_1, will be as follows:

$dice = \{d_1, d_2\}.\ \ score = \{1,2,3,4,5,6\}.\ \ person = \{mike, john\}.$
$roll : dice \rightarrow score.$
rel $owns : dice \times person,\ even : dice,\ even.$

The **regular part** of a P-log program consists of a collection of rules of $\mathcal{L}$. A rule can contain atoms of the form $a(\bar{t}) = y$ which are viewed as shorthand for an $\mathcal{L}$ atom $a(\bar{t}, y)$). For instance, regular part D_2 of program Π_0 may contain rules of A-Prolog

$even(D) \leftarrow roll(D) = Y, Y\ mod\ 2 = 0.$
$\neg even(D) \leftarrow not\ even(D).$
$even \leftarrow roll(d_1) = Y_1, roll(d_2) = Y_2, (Y_1 + Y_2)\ mod\ 2 = 0.$
$owns(d_1, mike).\ \ owns(d_2, john).$

Probabilistic information consist of statements of the form:

$$random\ \ a(\bar{t})\ :\ B \tag{3}$$

$$pr(a(\bar{t}) = y\ |_c\ B) = v \tag{4}$$

where $v \in [0,1]$, B is a collection of Σ-literals, and pr is a special symbol not belonging to Σ. By $pr(a(\bar{t}) = y\ |_c\ B)$ we denote the probability of $a(\bar{t}) = y$ *being caused by factors determined by* B. If B is empty we simply write $pr(a(\bar{t}) = y)$. (3) says that, given B, the value of $a(\bar{t})$ is normally selected at random; (4) gives a causal probability of a particular selection. For instance, the dice domain may include probabilistic part, D_3:

$random\ \ roll(D).$
$pr(roll(D) = Y\ |_c\ owns(D, john)) = 1/6.$
$pr(roll(D) = 6\ |_c\ owns(D, mike)) = 1/4.$
$pr(roll(D) = Y\ |_c\ Y \neq 6, owns(D, mike)) = 3/20.$

This says that the die owned by John is fair, while the die owned by Mike is biased to roll 6 at a probability of .25. Statements of type (4) will be sometimes referred to as *probabilistic atoms*.

We will have a special agreement for boolean attributes. First, $pr(a(\bar{t}) = true)$ and $pr(a(\bar{t}) = false)$ will be written as $pr(a(\bar{t}))$ and $pr(\neg a(\bar{t}))$. Second, for each probabilistic atom $pr(a(\bar{t})) = v$ from the program we will automatically generate the atom $pr(\neg a(\bar{t})) = 1 - v$. This will allow the user to write fewer probabilistic atoms.

Observations and **actions** are statements of the respective forms

$$obs(l). \qquad do(l).$$

Observations are used to record the outcomes of random events. The dice domain may, for instance, contain $\{obs(roll(d_1) = 4)\}$ recording the outcome of rolling dice d_1. $do(l)$ indicates that l is made true as a result of a deliberate (non-random) action. For instance, $\{do(roll(d_1) = 4)\}$ may indicate that d_1 was simply put on the table in the described

position. The meaning of do is briefly discussed in the definition of the semantics and in Examples 3 and 4. For more detailed discussion of the difference between actions and observations see [15]. The program Π_0 obtained from Π by removing observations and actions will be referred to as the *base* of Π.

2.2 Semantics of P-log

The semantics of a probabilistic program Π (over dialect $\mathcal{L}$ of A-Prolog) is given by the sets of beliefs of a rational agent associated with Π, together with their probabilities. Sometimes we refer to these sets as possible worlds of Π. *Formulas* of Π are constructed from atoms and the symbol $true$ using $\wedge$, or , and $\neg$. The semantics of P-log is based on the following steps:

1. The P-log program Π is mapped to a program Π' of $\mathcal{L}$ (see below).
2. The set, W, of Σ-literals from an answer set of Π' is viewed as a *possible world* (answer set) of Π. W can be viewed as a partial interpretation of a formula F which can be true, false, or undefined in W.
3. The *unnormalized probability*, $\hat{P}_{\Pi}(W)$, of a possible world W is

$$\hat{P}_{\Pi}(W) = \prod_{pr(l,v)\in W} v$$

4. The *probability of a formula* A, $P_{\Pi}(A)$, is defined as the sum of the unnormalized probabilities of the possible worlds of Π satisfying A divided by the sum of the unnormalized probabilities of all possible worlds of Π. We refer to P_{Π} as the *probability measure* defined by Π.
5. The *conditional probability*, $P_{\Pi}(A|B)$, is defined as the probability $P_R(A)$ where $R = \Pi \cup \{obs(B)\}$. (See Proposition 2 for the relationship between this notion and the usual definition).

Π' is a program of $\mathcal{L}$ consisting of sort declarations of Π (with $c = \{x_1, \ldots, x_n\}$ interpreted as $c(x_1), \ldots, c(x_n)$), its regular part, actions and observations, and the collection of rules (5)-(12).

• For each non-boolean attribute a with range $\{y_1, \ldots, y_m\}$:

$$a(\overline{X}, y_1) \text{ or } \ldots \text{ or } a(\overline{X}, y_m) \leftarrow random(a(\overline{X})) \tag{5}$$

• For each boolean attribute a:

$$a(\overline{X}) \text{ or } \neg a(\overline{X}) \leftarrow random(a(\overline{X})) \tag{6}$$

(Note that in both cases $\overline{X}$ will not be present for attributes of arity 0).

• For each attribute a:

$$\neg a(\overline{X}, Y_1) \leftarrow a(\overline{X}, Y_2), Y_1 \neq Y_2, \tag{7}$$

The rules (5),(6),(7) imitate random selection of the values of a.

• For each declaration (3):

$$random(a(\bar{t})) \leftarrow B, not\ \neg random(a(\bar{t})) \tag{8}$$

This rule captures the meaning of "normally" in the informal interpretation of (3).

• Cancellation axiom for (8) for every attribute a:

$$\neg random(a(\overline{X})) \leftarrow do(a(\overline{X}) = Y) \tag{9}$$

This axiom (along with (11)) captures the meaning of the do statement: the value of $a(\overline{X})$ is not random if it is selected by a deliberate action.

• For each probability atom (4):

$$pr(a(\bar{t}, y), v) \leftarrow B, a(\bar{t}, y), random(a(\bar{t})). \tag{10}$$

This rule assigns probability v to $a(\bar{t}) = y$ in every possible world in which $a(\bar{t}) = y$ is caused by B.

• Observations and deliberate actions: For every attribute a,

$$\leftarrow obs(a(\overline{X}, Y)), not\ a(\overline{X}, Y)). \quad a(\overline{X}, Y)) \leftarrow do(a(\overline{X}, Y)). \tag{11}$$

These rules are used to make sure that the program's beliefs match the reality. Note that the latter establishes the truth of l while the former only eliminates models not containing l.

• Eliminating impossible worlds:

$$\leftarrow pr(a(\overline{X}, Y), 0). \tag{12}$$

This rule ensures that every possible world of the program truly possible, i.e., has a non-zero probability. This completes the construction of Π'.

Definition 1. A probabilistic program Π is said to be *consistent* if

1. Π' is consistent (i.e., has a consistent answer set).
2. Let Π_0 be the base of Π. Then, for any probability atom $pr(l|_c B) = y$ from Π_0, the conditional probability $P_{\Pi_0}(l|B) = y$ whenever the latter is defined.
3. Whenever $pr(l|B_1) = y_1$ and $pr(l|B_2) = y_2$ belong to Π, no possible world of Π satisfies B_1 and B_2.

The first requirement ensures the consistency of the program rules. The second guarantees that P_Π satisfies the probabilistic statements from Π. The third requirement enforces the independence assumptions embodied in causal probabilities: given its cause B, an effect l has a fixed probability, independent of all other factors (except for effects of l).

The following proposition says that P_Π satisfies axioms of probability.

Proposition 1. For consistent P-log program Π:

1. For every formula A, $0 \leq P_\Pi(A) \leq 1$,

2. $P_{\Pi}(\ \mathit{true}\ ) = 1$, and
3. $P_{\Pi}(A \text{ or } B) = P_{\Pi}(A) + P_{\Pi}(B)$, for any mutually exclusive formulas A and B.

(Note that, since A or $\neg A$ may be undefined in a possible world W, $P_{\Pi}(A \text{ or } \neg A)$ is not necessarily equal to 1).

To illustrate these definitions let us further elaborate the "dice" example.

Example 2. Let T_0 consist of first three sections D_1, D_2, D_3, of the "dice" program from Example 1. Then T_0' consists of rules of D_2 and the rules:

$dice(d1).\ dice(d2).\ person(mike).\ person(john).$
$score(1).\ score(2).\ score(3).\ score(4).\ score(5).\ score(6).$
$roll(D,1)\ or\ roll(D,2)\ or\ \ldots\ or\ roll(D,5)\ or\ roll(D,6) \leftarrow random(roll(D)).$
$\neg roll(D,Y_2) \leftarrow roll(D,Y_1), Y_1 \neq Y_2.$
$random(roll(D)) \leftarrow not\ \neg random(roll(D)).$
$\neg random(roll(D)) \leftarrow do(roll(D) = Y).$
$pr(roll(D,Y),1/6) \leftarrow owns(D,john), roll(D,Y), random(roll(D)).$
$pr(roll(D,6),1/4) \leftarrow owns(D,mike), roll(D,6), random(roll(D)).$
$pr(roll(D,Y),3/20) \leftarrow Y \neq 6, owns(D,mike), roll(D,Y), random(roll(D)).$
$\leftarrow obs(a(X,Y)), not\ a(X,Y).$
$a(X,Y) \leftarrow do(a(X,Y).$
$\leftarrow pr(a(X,Y),0).$

It is easy to check that T_0' has 36 answer sets containing different pairs of atoms $roll(d_1,i_1)$ and $roll(d_2,i_2)$. Each answer set of T_0' containing $roll(d_1,6)$ will contain a probability atom $pr(roll(d_1,6),1/4)$, as well as a probability atom $pr(roll(d_2,i),1/6)$ for some i, and hence have the probability $1/24$. Any other answer set has probability $1/40$. It is easy to check that the program is consistent.

Now let $T_1 = T_0 \cup \{obs(roll(d_1,4))\}$. By definition,
$P_{T_0}(even|roll(d_1,4)) = P_{T_1}(even) = 1/2$. The same result can be obtained by using classical definition of conditional probability,

$$P(A|B) = P(A \wedge B)/P(B) \tag{13}$$

The following proposition shows that this is not a coincidence. A dialect $\mathcal{L}$ of A-Prolog is called *monotonic with respect to constraints* if for every program Π and constraint $\leftarrow B$ of $\mathcal{L}$ any answer set of $\Pi \cup \{\leftarrow B\}$ is also an answer set of Π.

Proposition 2. Let $\mathcal{L}$ be a dialect of A-Prolog monotonic with respect to constraints and let Π be a consistent P-log program over $\mathcal{L}$. Then for every A and every B with $P_{\Pi}(B) \neq 0$, P_{Π} satisfies condition (13) above.

Example 3. Consider a program, P_0

$random\ a : boolean.$
$pr(a) = 1.$

Recall that P_0 will be (automatically) expanded to include a new probability atom, $pr(\neg a) = 0$. It is easy to see that P_0' has one answer set, which contains a (the possible

answer set containing $\neg a$ is eliminated by constraint (12)). Obviously, $P_{P_0}(a) = 1$ and hence the program is consistent. Now we compare P_0 with the following program P_1:

$random\ a : boolean.$
$a.$

The programs have the same possible worlds and the same probability measures. However, they express different information. To see that, consider programs P_2 and P_3 obtained by adding the statement $do(\neg a)$ to P_0 and P_1 respectively. P_2 remains consistent — it has one possible world $\{\neg a\}$ — while P_3 becomes inconsistent (see rule (11)). The statement $pr(a) = 1$ is defeasible while the statement a is not. This does not mean however that the former can be simply replaced by the corresponding default.

Consider Π_4

$random\ a : boolean.$
$a \leftarrow not\ \neg a.$

Π_4 has two possible worlds, $\{a\}$ and $\{\neg a\}$ (note the interplay between the default and rule 6 of Π_4'). In other words "randomness" undermines the default.

Finally consider P_5:

$random\ a : boolean.$
$a.$
$pr(a) = 1/2.$

and P_6:

$random\ a : \{0, 1, 2\}.$
$pr(a = 0) = pr(a = 1) = pr(a = 2) = 1/2.$

Both programs are inconsistent. P_5 has one possible world $W = \{a\}$. $\hat{P}_{P_5}(W) = 1/2$, $P_{P_5}(a) = 1$ instead of $1/2$.

P_6 has three possible worlds, $\{a(0), \neg a(1), \neg a(2)\}$, $\{\neg a(0), a(1), \neg a(2)\}$, and $\{\neg a(0), \neg a(1), a(2)\}$ each with unnormalized probability $1/2$. Hence $P_{P_6}(a(0)) = 1/3$ instead of $1/2$. (Let $V(B, \bar{t})$ be a multiset of v such that $pr(a(\bar{t} = y) = v \in \Pi$ for some $y \in range(a)$. Then it can be shown that if Π is consistent then for every B and $\bar{t}$ the sum of the values in $V(B, \bar{t})$ is 1).

3 Representing Knowledge in P-log

Now we give several examples of non-trivial probabilistic knowledge representation and reasoning performed in P-log.

Example 4. (Simpson's Paradox)
Let us consider the following story from [15]: A patient is thinking about trying an experimental drug and decides to consult a doctor. The doctor has tables of the recovery rates that have been observed among males and females, taking and not taking the drug.

Males:	recover	-recover	num_of_people	recovery_rate
drug	18	12	30	60%
-drug	7	3	10	70%

Females:	recover	-recover	num_of_people	recovery_rate
drug	2	8	10	20%
-drug	9	21	30	30%

What should the doctor's advice be? Assuming that the patient is a male, the doctor may attempt to reduce the problem to checking the following inequality

$$P(recover|male, drug) \ > \ P(recover|male, \neg drug) \tag{14}$$

The corresponding probabilities, given by the tables, are 0.6 and 0.7. The inequality fails, and hence the advice is not to take the drug. This, indeed, is the correct advice. A similar argument shows that a female patient should not take the drug.

But what should the doctor do if he forgotten to ask the patient's sex? Following the same reasoning, the doctor might check whether

$$P(recover|drug) \ > \ P(recover|\neg drug) \tag{15}$$

This will lead to an unexpected result. $P(recovery|drug) = 0.5$ while $P(recovery|\neg drug) = 0.4$. The drug seems to be beneficial to patients of unknown sex — though similar reasoning has shown that the drug is harmful to the patients of known sex, whether they are male or female!

This phenomenon is known as Simpson's Paradox: conditioning on A may increase the probability of B among the general population, while decreasing the probability of B in every subpopulation (or vice-versa). In the current context, the important and perhaps surprising lesson is that conditional probabilities do not faithfully formalize what we really want to know: *what will happen if we do X?* In [15] Pearl suggests a solution to this problem in which the effect of action A on condition C is represented by $P(C|do(A))$ — a quantity defined in terms of graphs describing causal relations between variables. Correct reasoning therefore should be based on evaluating the inequality

$$P(recover|do(drug)) > P(recover|do(\neg drug)) \tag{16}$$

instead of (15) (similarly for (14)). In Pearl's calculus the first value equals $.4$, the second, $.5$. The drug is harmful for the general population as well.

Note that in our formalism $P_{\Pi}(C|do(A)$ is defined simply as $P_R(C)$ where $R = \Pi \cup \{do(A)\}$ and hence P-log allows us to directly represent this type of reasoning. We follow [15] and assume that the tables, together with our intuition about the direction of causality between the variables, provide us with the values of the following causal probabilities:

$pr(male) = 0.5$, $pr(recover|_c male, drug) = 0.6$,
$pr(recover|_c male, \neg drug) = 0.7$, $pr(recover|_c \neg male, drug) = 0.2$,

$pr(recover|_c \neg male, \neg drug) = 0.3$, $pr(drug|_c male) = 0.75$, $pr(drug|_c \neg male) = .25$.

These statements, together with declarations:

$random\ \ male, recover, drug\ :\ boolean$

constitute a probabilistic logic program, Π, formalizing the story. The program describes eight possible worlds containing various values of the attributes. Each world is assigned a proper probability value, e.g. $P_\Pi(\{male, recover, drug\}) = .5 * .6 * .75 = 0.225$. It is not difficult to check that the program is consistent. The values of $P_\Pi(recover|_c do(drug)) = .4$ and $P_\Pi(recover|_c do(\neg drug)) = .5$ can be computed by finding $P_{\Pi_1}(recover)$ and $P_{\Pi_2}(recover)$, where $\Pi_1 = \Pi \cup \{do(drug).\}$ and $\Pi_2 = \Pi \cup \{do(\neg drug).\}$.

Now we consider several reasoning problems associated with the behavior of a malfunctioning robot. The original version, not containing probabilistic reasoning, first appeared in [6] where the authors discuss the difficulties of solving the problem in Situation Calculus.

Example 5. (A malfunctioning robot)

There are rooms, r_0, r_1, and r_2, reachable from the current position of a robot. The robot navigation is usually successful. However, a malfunction can cause the robot to go off course and enter any one of the rooms. The doors to the rooms can be open or closed. The robot cannot open the doors.

The authors want to be able to use the corresponding formalization for correctly answering simple questions about the robot's behavior including the following "typical" scenario: The robot moved toward open room r_1 but found itself in some other room. What room can this be?

The initial story contains no probabilistic information so we start with formalizing this knowledge in A-Prolog. First we need sorts for time-steps and rooms. (Initial and final moments of time suffice for our purpose).

$time = \{0, 1\}$. $rooms = \{r_0, r_1, r_2\}$.

In what follows we use variable T for time and R for rooms. There will be two actions:

$enter(T, R)$ - the robot *attempts* to enter the room R at time step T.

$break(T)$ - an exogenous breaking action which may alter the outcome of this attempt.

A state of the domain is modeled by two time-dependent relations $open(R, T)$ (room R is opened at moment T), $broken(T)$ (robot is malfunctioning at T), and the attribute, $in(T) : time \rightarrow rooms$, which gives the location of the robot at T.

The description of dynamic behavior of the system is given by A-Prolog rules:

Dynamic causal laws describe direct effects of the actions (note that the last law is non-deterministic):

$broken(T + 1) \leftarrow break(T)$.

$in(T+1,R) \leftarrow enter(T,R), \neg broken(T+1).$

$in(T+1,r_0)$ or $in(T+1,r_1)$ or $in(T+1,r_2) \leftarrow broken(T), enter(T,R).$

To specify that the robot cannot go through the closed doors we use a constraint:

$\leftarrow \neg in(T,R), in(T+1,R), \neg open(R,T).$

Moreover, the robot will not even attempt to enter the room if its door is closed.

$\leftarrow enter(T,R), \neg open(R,T).$

To indicate that in is a function we use static causal law:

$\neg in(T,R_2) \leftarrow in(T,R_1), R_1 \neq R_2.$

We also need the inertia axiom:

$in(T+1,R) \leftarrow in(T,R), not\ \neg in(T+1,R).$
$broken(T+1) \leftarrow broken(T), not\ \neg broken(T).$
$\neg broken(T+1) \leftarrow \neg broken(T), not\ broken(T).$
(Similarly for $open$).

Finally, we describe the initial situation:

$open(R,0) \leftarrow not\ \neg open(R,0).$
$in(0,r_1).$
$\neg in(0,R) \leftarrow not\ in(0,R).$
$\neg broken(T) \leftarrow not\ broken(T).$

The resulting program, Π_0, completes the first stage of our formalization.

It is easy to check that $\Pi_0 \cup \{enter(0,r_0)\}$ has one answer set, A, and that $in(1,r_0) \in A$. Program $\Pi_0 \cup \{enter(0,r_0), break(0)\}$ has three answer sets containing $in(1,r_0)$, $in(1,r_1)$, and $in(1,r_2)$ respectively. If, in addition, we are given $\neg open(r_2,0)$ the third possibility will disappear.

Now we show how this program can be extended by probabilistic information and how this information can be used together with regular A-Prolog reasoning.

Consider Π_1 obtained from Π_0 by adding

random $in(T+1) : enter(T,R), broken(T+1).$
$pr(in(T+1,R)|_c\ enter(T,R), broken(T+1)) = 1/2.$
$pr(in(T+1,R_1)|_c\ R_1 \neq R_2, enter(T,R_2), broken(T+1)) = 1/4.$

together with the corresponding declarations, e.g.

$in : time \rightarrow rooms.$

It is not difficult to check that probabilistic program $T_1 = \Pi_1 \cup \{enter(0,r_0)\}$ has the unique possible world which contains $in(1,r_0)$. Hence, $P_{T_1}(in(1,r_0)) = 1$. It is easy to show that Π_1 is consistent. (Note that the conditional probabilities corresponding to the probability atoms of Π_1, e.g., $P_{T_1}(in(1,r_0)|broken(1))$, are undefined and hence (2) of the definition of consistency is satisfied.)

The program $T_2 = T_1 \cup \{break(0)\}$ has three possible worlds — A_0 containing $in(1, r_0)$, and A_1, A_2 containing $in(1, r_1)$ and $in(1, r_2)$ respectively; $P_{T_2}(A_0) = 1/2$ while $P_{T_2}(A_1) = P_{T_2}(A_2) = 1/4$. It is easy to see that T_2 is consistent. Note that $P_{T_1}(in(1, r_0)) = 1$ while $P_{T_2}(in(1, r_0)) = 1/2$ and hence the *additional information changed the degree of reasoner's belief.*

So far our probabilistic programs were based on A-Prolog. The next example shows the use of P-log programs over CR-Prolog [2] — an extension of A-Prolog which combines regular answer set reasoning with abduction. In addition to regular rules of A-Prolog the new language allows so called *consistency-restoring* rules, i.e., rules of the form

$$l \stackrel{+}{\leftarrow} B.$$

which say that, given B, l may be true but this is a rare event which can be ignored unless l is needed to restore consistency of the program. The next example elaborates the initial formalization of the robot story in CR-Prolog.

Example 6. (Probabilistic programs over CR-Prolog)
Let us expand the program T_1 from Example 5 by a CR-rule

$$break(T) \stackrel{+}{\leftarrow} \tag{17}$$

The rule says that even though the malfunctioning is rare it may happen. Denote the new program by T_3.

The semantics of CR-Prolog guarantees that for any collection I of atoms such that $T_1 \cup I$ is consistent, programs $T_1 \cup I$ and $T_3 \cup I$ have the same answer sets; i.e., the conclusions we made so far about the domain will not change if we use T_3 instead of T_1. The added power of T_3 will be seen when the use of T_1 leads to inconsistency. Consider for instance the scenario $I_0 = \{obs(\neg in(1, r_0))\}$. The first formalization could not deal with this situation — the corresponding program would be inconsistent.

The program $T_4 = T_3 \cup I_0$ will use the CR-rule (17) to conclude $break(0)$, which can be viewed as a diagnosis for an unexpected observation. T_4 has two answer sets containing $in(1, r_1)$ and $in(1, r_2)$ respectively. It is not difficult to check that $P_{T_3}(in(1, r_0)) = 1$ while $P_{T_3}(in(1, r_0)|I_0) = P_{T_4}(in(1, r_0)) = 0$. Interestingly, *this phenomenon cannot be modeled using classical conditional probabilities*, since classically whenever $P(A) = 1$, the value of $P(A|B)$ is either 1 or undefined.

Our last example will show how Π_2 can be modified to introduce some additional probabilistic information and used to obtain most likely diagnoses.

Example 7. (Doing the diagnostics)
Suppose we are given a list of mutually exclusive faults which could be caused by the breaking action, together with the probabilities of these faults. This information can be incorporated in our program, Π_2, by adding $faults = \{f_0, f_1\}$. $fault : time \rightarrow faults$.
$random\ fault(T + 1) : break(T)$.
$pr(fault(T, f_0)|_c broken(T)) = .4 \quad pr(fault(T, f_1)|_c broken(T)) = .6$ Let us also

assume that chances of the malfunctioning robot to get to room R are determined by the type of the faults, e.g. $pr(in(1,r_0)|_c fault(1,f_0)) = .2\ pr(in(1,r_0)|_c fault(1,f_1) = .1$

$pr(in(1,r_1)|_c fault(1,f_0)) = .6 \quad pr(in(1,r_1)|_c fault(1,f_1)) = .5$
$pr(in(1,r_2)|_c fault(1,f_0)) = .2 \quad pr(in(1,r_2)|_c fault(1,f_1)) = .4$

Note that this information supersedes our previous knowledge about the probabilities of in and hence should replace the probabilistic atoms of Π_2. The resulting program, Π_3, used together with $\{enter(0,r_0), obs(\neg in(0,r_0))\}$ has four answer sets weighted by probabilities. Simple computation shows at moment 1 the robot is most likely to be in room r_1.

4 Relationship to Existing Work

Our work was greatly influenced by J. Pearl's view on causality and probability. It can be shown that the Bayesian Networks and Probabilistic Causal Models of Pearl can be mapped into P-log programs of similar size. (Proofs of the corresponding theorems will be given in the full version of this paper.) The examples discussed above show that, in addition, P-log allows natural combination of logical and probabilistic information. We were influenced to a lesser degree, by various work incorporating probability in logic programming [11,9,7,10,8]. In part this is due to our use of answer set semantics, which introduces unique challenges (as well as benefits, in our opinion) for the integration of probabilities.

The closest to our approach is that of Poole [18,17]. We note three major differences between our work and the work of Poole [18,17]. First, A-Prolog provides a richer logical framework than does choice logic, including default and classical negation and disjunction. Moreover, our approach works, without modification, with various extensions of A-Prolog including the use of CR-rules. Second, in contrast to Poole's system, the logical aspects of P-log do not "ride on top" of the mechanism for generating probabilities: we bring to bear the power of answer set programming, not only in describing the consequences of random events, but also in the description of the underlying probabilistic mechanisms. Third, our formalization allows the distinction between observations and actions (i.e., doing) to be expressed in a natural way, which is not addressed in choice logic.

There are three elements which, to our knowledge, are new to this work. First, rather than using classical probability spaces in the semantics of P-log, we define probabilities of formulas directly in terms of the answer set semantics. In this way, A P-log program *induces* a classical probability measure on possible worlds by its construction, rather than relying on the existence of a classical measure compatible with it. We see several advantages to our re-definition of probabilities. Most notably, the definition of conditional probability becomes more natural, as well as more general (see Example 6). Also, possible worlds and events correspond more intuitively to answer sets and formulas than to the sample points and random events (i.e., sets of sample points) of the classical theory.

Second, P-log allows us to *elaborate on defaults by adding probabilities* as in Examples 6-7. Preferences among explanations, in the form of defaults, are often more easily

available from domain experts than are numerical probabilities. In some cases, we may want to move from the former to the latter as we acquire more information. P-log allows us to represent defaults, and later integrate numerical probabilities by adding to our existing program rather than modifying it. Finally, the semantics of P-log over CR-Prolog gives rise to a unique phenomenon: we can move from one classical probability measure to another merely by adding observations to our knowledge base, as in Example 6. This implies that P-log probability measures are more general than classical ones, since the measure associated with a single P-log program can, through conditioning, address situations that would require multiple distinct probability spaces in the classical setup.

References

1. F. Bacchus. Representing and reasoning with uncertain knowledge. MIT Press, 1990
2. M. Balduccini and M. Gelfond. Logic Programs with Consistency-Restoring Rules. In AAAI Spring 2003 Symposium, 2003.
3. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem solving in dlv. In J. Minker, editor, *Logic Based Artificial Intelligence*, pages 79–103. Kluwer Academic publisher, 2000.
4. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. In *New Generation Computing*, 365–387, 1991.
5. L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Relational data mining*, pages 307–335. Springer, 2001.
6. G. Iwan and G. Lakemeyer. What observations really tell us. In *CogRob'02*, 2002.
7. Kristian Kersting and Luc De Raedt. Bayesian logic programs. In J. Cussens and A. Frisch, editors, *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, pages 138–155, 2000.
8. T. Lukasiewicz. Probabilistic logic programming. In *ECAI*, pages 388–392, 1998.
9. S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, page 29. Department of Computer Science, Katholieke Universiteit Leuven, 1995.
10. Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.
11. Liem Ngo and Peter Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171(1–2):147–177, 1997.
12. I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proc. 4th international conference on Logic programming and non-monotonic reasoning*, pages 420–429. Springer, 1997.
13. P. Simons,I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence Journal*,138:181–234, 2002/
14. J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, 1988.
15. J. Pearl. *Causality*. Cambridge University Press, 2000.
16. D. Poole. The independent choice logic for modeling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2 (special issue on economic principles of multi-agent systems)):7–56, 1997.
17. D. Poole. Abducing through negation as failure: Stable models within the independent choice logic. *Journal of Logic Programming*, 44:5–35, 2000.
18. David Poole. Probabilistic horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.

Answer Sets: From Constraint Programming Towards Qualitative Optimization

Gerhard Brewka

University of Leipzig
Dept. of Computer Science
Augustusplatz 10-11
04109 Leipzig, Germany
brewka@informatik.uni-leipzig.de

Abstract. One of the major reasons for the success of answer set programming in recent years was the shift from a theorem proving to a constraint programming view: problems are represented such that stable models, respectively answer sets, rather than theorems correspond to solutions. This shift in perspective proved extremely fruitful in many areas. We believe that going one step further from a "hard" to a "soft" constraint programming paradigm, or, in other words, to a paradigm of qualitative optimization, will prove equally fruitful. In this paper we try to support this claim by showing that several generic problems in logic based problem solving can be understood as qualitative optimization problems, and that these problems have simple and elegant formulations given adequate optimization constructs in the knowledge representation language.

1 Introduction

Answer sets [17], originally invented to define the semantics of (extended) logic programs with default negation, have proven to be extremely useful for solving a large variety of AI problems. Two important developments were essential for this success:

1. the development of highly efficient answer-set provers, the most advanced among them being *Smodels* [21] and *dlv* [12];
2. a shift from a theorem proving to a constraint programming perspective [20],[19].

Let us focus on the latter here. It turned out that many problems, for instance in reasoning about actions, planning, diagnosis, belief revision and product configuration, have elegant formulations as logic programs so that models of programs, rather than proofs of queries, describe problem solutions [18,28,1,13]. This view of logic programs as constraints on the sets of literals which may count as solutions has led to a new problem solving paradigm called answer set programming (ASP).

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 34–46, 2004.

The major goal of this paper is to push this shift in perspective one step further: from a constraint programming paradigm to a paradigm of qualitative optimization. Many problems have natural formulations as optimization problems, and many problems which can be represented in a "hard" constraint programming paradigm have fruitful, more flexible refinements as optimization problems.

Consider planning as a typical example. We know how to represent planning problems as ASP problems. Moving from a constraint programming to an optimization perspective allows us to specify criteria by which we can rank plans according to their quality. This allows us to select good plans (or to generate suboptimal plans if there is no way to satisfy all requirements). An example for quantitative optimization is planning under action costs [15].

Our interest in optimization based on qualitative preferences stems from the fact that for a variety of applications numerical information is hard to obtain (preference elicitation is rather difficult) - and often turns out to be unnecessary. This paper contains several examples of this kind.

Of course, the use of optimization techniques in answer set programing is not new. There is a large body of work on preferred answer sets, see for instance [5, 26] and the references in these papers. Also some of the existing solvers have (numerical) optimization facilities: *Smodels* with weight constraints has *maximize* and *minimize* statements operating on weights of atoms in answer sets [27]. An interesting application of these constructs to modeling auctions can be found in [2]. *dlv* has weak constraints [8] of the form

$$\leftarrow a_1, \ldots a_n, not\ b_1, \ldots, not\ b_m.\ [w : l]$$

where w is a numerical penalty and l is a priority level. For each priority level, the sum of the penalties of all violated constraints (i.e., constraints whose bodies are satisfied) is computed. The answer sets with minimal overall penalty in one level are compared based on the overall penalties of the next level, etc. Such constaints were used, for instance, to implement planning under action costs as described in [15] and for knowledge based information-site selection [14].

However, a systematic treatment of answer set programming as *qualitative optimization* is still lacking. This paper is meant as a step into this direction. The major goal is to substantiate - by means of several generic examples - the claim that optimization techniques, and in particular qualitative techniques, are highly fruitful in a large number of applications.

This paper will not introduce any new formalism. Instead, it will use existing formalisms, co-developed by the author, to demonstrate how these formalisms can be used fruitfully for a variety of problems. We will use logic programs with ordered disjunction (LPODs) as described in [4] and [7], as well as the framework of optimization programs as described in [6]. Both have their merits, the second being more modular and general. However, in some cases the former admits more concise representations, as we will see.

The problems we describe in this paper are generic in the sense that they may arise in a number of different applications. We will first discuss the use of

optimization techniques for abduction, that is, the generation of explanations for observed behaviour, and diagnosis. The major goal here is to give an example showing that it is indeed extremely helpful to have optimization techniques available as part of the semantics of the underlying formalism. We will give straightforward translations of abduction and also of consistency based diagnosis techniques to LPODs. Our translations are influenced by [13]. However, due to the availability of optimization constructs in the representation language our translations are considerably simpler than those used for the *dlv* diagnosis frontend.

We next discuss the problem of solution coherence. Consider a domain where solutions need to be computed ahead of time, but later on new constraints may have to be taken into account. A typical example (which was discussed at the Banff Workshop on Constraint Programming, Belief Revision and Combinatorial Optimization, May 03) is the scheduling of meetings in an organization. Persons who need to attend a particular meeting must be informed about this fact early enough. On the other hand, it may always happen that, due to unforeseen circumstances, new "hard" constraints arise. Now assume an automated scheduling system based on answer set programming were to recompute a new solution based on the new set of constraints from scratch. This could lead to a new solution of the scheduling problem with completely different time slots for a large number of meetings. This is certainly unwanted: what we would like to have is a new solution which is as close as possible to the original solution. This is obviously a qualitative optimization problem and we will show how this problem can be handled using optimization programs.

Finally, we will show that inconsistency handling can be viewed as an optimization problem. The basic idea is to re-represent the original information in such a way that rules can be arbitrarily disregarded. We then use preferences expressing that rules should not be disregarded. We can go further and add meta preferences between these preferences to distinguish between rules which should not be disregarded with higher priority. We can also introduce contradiction removal rules [23] which are applied only in case an inconsistency arises.

From the discussion of the examples we derive a number of requirements for a general specification language for qualitative optimization. Such a language should allow us to express preferences of a very general kind together with different combination strategies, and it should be possible to apply different strategies to different parts of the preference program. We strongly believe that extending answer set programming in this direction will greatly increase its impact as a new declarative programming paradigm.

We assume the reader is familiar with the notion of answer sets [17]. A short introduction to LPODs and optimization programs is given in the next section.

2 LPODs And Optimization Programs

In this section we briefly recall the basic notions underlying the formalisms to be used in this paper, namely LPODs [4,7] and optimization programs [6]. LPODs

use ordered disjunction $\times$ in the head of rules to express preferences among literals in the head: the rule $r = A_1 \times \ldots \times A_n \leftarrow body$ says: if *body* is satisfied then some A_i must be in the answer set, most preferably A_1, if this is impossible then A_2, etc. Answer sets are defined through split programs [25] containing exactly one option for each of the original LPOD rules, where, for $k \leq n$, option r^k of the rule above is $A_k \leftarrow body, not\ A_1, \ldots, not\ A_{k-1}$.

An answer set S can satisfy rules like r to different degrees, where smaller degrees are better: if *body* is satisfied, then the satisfaction degree is the smallest index i such that $A_i \in S$. Otherwise, the rule is irrelevant. Since there is no reason to blame an answer set for not applying an inapplicable rule we also define the degree to be 1 in this case. The degree of r in S is denoted $deg_S(r)$.

Based on the satisfaction degrees of single rules a global preference ordering on answer sets is defined. This can be done through a number of different combination strategies. Let $S^k(P) = \{r \in P \mid deg_S(r) = k\}$. For instance, we can use one of the following conditions to define that S_1 is strictly preferred to S_2:

1. there is a rule satisfied better in S_1 than in S_2, and no rule is satisfied better in S_2 than in S_1 (Pareto);
2. at the smallest degree j such that $S_1^j(P) \neq S_2^j(P)$ we have $S_1^j(P) \supset S_2^j(P)$ (inclusion);
3. at the smallest degree j such that $|S_1^j(P)| \neq |S_2^j(P)|$ we have $|S_1^j(P)| > |S_2^j(P)|$ (cardinality);
4. the sum of the satisfaction degrees of all rules is smaller in S_1 than in S_2 (penalty sum).

Note that S_1 is inclusion preferred to S_2 whenever it is Pareto preferred to S_2, and cardinality preferred to S_2 whenever it is inclusion preferred to S_2. The penalty sum strategy can be used to model *dlv*'s weak constraints, provided all weights are integers: a weak constraint $\leftarrow body$ with weight n can be represented as $\bot \times \ldots \times \bot \times \top \leftarrow body$, where the ordered disjunction has n disjuncts.

In the LPOD approach the construction of answer sets is amalgamated with the expression of preferences. Optimization programs, on the other hand, separate the two consequently to allow for greater modularity, flexibility and generality. An optimization program is a pair (P_{gen}, P_{pref}) consisting of an arbitrary program P_{gen} used to generate answer sets, that is, P_{gen} can be any type of program as long as it produces sets of literals, and a preference program P_{pref}. The latter consists of rules of the form $C_1 > \ldots > C_n \leftarrow body$ where the C_i are boolean combinations of literals built from $\wedge, \vee, \neg$ and *not* . Again, if *body* is satisfied in an answer set S and some boolean combination C_i is satisfied, then the degree of the rule is the smallest such index. Otherwise the rule is irrelevant. Again we do not want to blame S for irrelevant rules and consider irrelevance as good a satisfaction degree as 1.

Again, the combination strategies described above - and many others - can be used to generate the global preference order. It is also possible to introduce meta preferences among the preference rules themselves by grouping them into subsets with different ranks. Preference rules with highest priority are considered

first. Intuitively, rules with lower priority are only used to distinguish between answer sets which are of the same quality according to more preferred rules. The reader is referred to the original papers for more details.

In the rest of this paper the inclusion-based combination strategy, that is, strategy 2 in the list above, will be our default strategy. As we will see, subset minimality is useful in many cases. Note that inclusion and Pareto preference coincide whenever all preference statements (ordered disjunctions or heads of preference rules) speak about no more than two options. Whenever we need cardinality-based preference we will mention this explicitly.

3 Abduction And Diagnosis

Abduction is the process of generating explanations for a set of observations. Most formal approaches to abduction are based on a fixed set of formulas H of possible explanations, a set of formulas K representing background knowledge, and a set of formulas O describing the observations to be explained [16,24,22,9]. An explanation then is a minimal subset H' of H satisfying:

1. $H' \cup K$ is consistent and,
2. $H' \cup K \models O$.

For the context of answer set programming let us assume that K is a logic program, H and O are sets of literals, and $\models$ is the credulous inference relation under answer set semantics, that is, for the program K and observations O we have $K \models O$ iff there is an answer set S of K such that $O \subseteq S$.[1] An explanation for O can be computed via the following LPOD $P_{abd}(K, H, O)$:

$$K \cup \{\leftarrow not\ o \mid o \in O\} \cup \{\neg ass(h) \times ass(h) \mid h \in H\} \cup \{h \leftarrow ass(h) \mid h \in H\}.$$

Intuitively, $ass(h)$ reads: h is assumed. The use of constraints for eliminating answer sets where some observation is unexplained, and the use of a predicate like ass is taken from [13]. However, the use of ordered disjunction is new and makes the representation considerably simpler. We have the following proposition:

Proposition 1. *Let H and O be sets of literals and let K be a logic program. H' is an explanation for O given K and H iff there is a consistent answer set S of $P_{abd}(K, H, O)$ such that $H' = \{h \in H \mid ass(h) \in S\}$.*

Abduction has a direct application in diagnosis. Diagnostic systems are an important class of knowledge based systems. Their main task is to determine reasons for observed malfunctioning. In the literature 2 kinds of approaches are usually distinguished:

– *abductive diagnosis:*
  a model M of the system to be diagnosed describes the effects of possible failures (diseases in the medical domain). Given a set of potential failures F

[1] This view is basically taken from [13]

and a set of observations O, an abductive diagnosis is an explanation of O from F with background knowledge M. In other words, abductive diagnosis is a special case of abduction where the hypotheses are potential failures and the background knowledge is the description of the system to be diagnosed.

– *consistency based diagnosis:*
here the model M describes the normal behaviour of the system, using abnormality predicates. A diagnosis is a minimal subset C' of the system components C such that assuming elements of C' to be abnormal and the others to be normal explains the observations. More formally, we are looking for a minimal subset C' of the components such that:
1. $\{ab(c) \mid c \in C'\} \cup \{\neg ab(c) \mid c \in C \setminus C'\} \cup M$ is consistent and,
2. $\{ab(c) \mid c \in C'\} \cup \{\neg ab(c) \mid c \in C \setminus C'\} \cup M \models O$.

Again we follow [13] in assuming that M is given as an (extended) logic program, F as a set of atoms, O as a set of literals, and that $\models$ is the credulous inference relation under answer set semantics, i.e. $P \models O$ says there is an answer set containing all literals in O. It is straightforward to express the conditions for both types of diagnosis using LPODs.

Abductive diagnosis is just a special case of abduction, and the program needed to compute diagnoses is just $P_{abd}(M, F, O)$ as described above.

For consistency based diagnosis we obtain the following program $P_{cd}(M, C, O)$:

$$M \cup \{\leftarrow \mathit{not}\ o \mid o \in O\} \cup \{\neg ab(c) \times ab(c) \in C\}.$$

Proposition 2. *Let C be a set of components, O a set of literals describing observations, and M a logic program describing the behavior of a system using the ab predicate. $C' \subseteq C$ is a consistency based diagnosis iff there is a consistent answer set S of $P_{cd}(M, C, O)$ such that $C' = \{c \mid ab(c) \in S\}$.*

Here is a small example due to [11]. Assume we have a battery connected in parallel to three bulbs. The program M consists of the facts:

$$\begin{array}{lll} \mathit{battery}(\mathit{bat}) & \mathit{bulb}(b_1) & \mathit{connected}(\mathit{bat}, b_1) \\ & \mathit{bulb}(b_2) & \mathit{connected}(\mathit{bat}, b_2) \\ & \mathit{bulb}(b_3) & \mathit{connected}(\mathit{bat}, b_3) \end{array}$$

together with the behaviour description:

$$\begin{array}{lll} \mathit{lit}(X) & \leftarrow & \mathit{bulb}(X), \mathit{not}\ ab(X), \\ & & \mathit{battery}(Y), \mathit{not}\ ab(Y), \mathit{connected}(Y, X) \\ \neg ab(X) & \leftarrow & \mathit{bulb}(X), \mathit{lit}(X) \\ \neg ab(Y) & \leftarrow & \mathit{bulb}(X), \mathit{lit}(X), \mathit{battery}(Y), \mathit{connected}(Y, X) \end{array}$$

Assume the observations are $O = \{\neg \mathit{lit}(b_1), \mathit{lit}(b_3)\}$, that is, we have to add the constraints:

$$\begin{array}{ll} \leftarrow & \mathit{not}\ \neg \mathit{lit}(b_1) \\ \leftarrow & \mathit{not}\ \mathit{lit}(b_3) \end{array}$$

Adding ordered disjunctions $\neg ab(c) \times ab(c)$ for $c \in \{bat, b_1, b_2, b_3\}$ we obtain a single preferred answer set containing $ab(b_1)$ and $\neg ab(c')$ for $c' \in \{bat, b_2, b_3\}$. The second answer set containing both $ab(b_1)$ and $ab(b_2)$ is non-preferred. The single diagnosis thus is b_1.

Eiter and colleagues also discuss single fault diagnosis [13]. Consistency based diagnosis can be restricted to the case where at most one component is faulty by adding the constraint $\leftarrow ab(X), ab(Y), X \neq Y$. However, in our case we can simply switch from an inclusion-based to a cardinality-based combination strategy if we want to minimize the number of components in a diagnosis. This has the advantage that we do not need to repeatedly replace the constraint with a weaker constraint allowing for more faults whenever a diagnosis with fewer faults does not exist.

It is illustrative to compare our simple formalizations with those in [13]. The lack of optimization facilities in the language used in that paper makes the translation of diagnosis problems to answer set programming considerably more difficult. There is a one page algorithm for the realization of subset minimality. We take this as support for our view that optimization facilities should be part of answer set programming languages.

4 Solution Coherence

In answer set programming problems are specified as programs and problem solutions correspond to answer sets of these programs. Assume we have computed a solution S to a problem specified through program P. Now consider some modification P' of P. At least in certain contexts it appears to be inappropriate to compute a new solution from scratch, that is, some arbitrary answer set S' of P'. We often want the new solution S' to be a minimal modification of S.

An illustrative example is a meeting scheduling system where P describes meetings, required participants, time restrictions of participants etc. and where S describes the scheduled meetings. If for one of the required participants of a particular meeting a new time constraint c arises, then S may no longer be a valid solution of $P' = P \cup \{c\}$. It is obvious that in this example we are not looking for some arbitrary solution of P', but for a solution of P' which is *as close as possible to* S.

Given a logic programming language with optimization facilities it is straightforward to represent this. What we need is a distance measure between answer sets. Here is one such measure based on symmetric differences. Let L, L_1 and L_2 be sets of literals. We say L_1 is closer to L than L_2, denoted $L_1 \leq_L L_2$, iff

$$\{a \mid a \in L \setminus L_1 \text{ or } a \in L_1 \setminus L\} \subseteq \{a \mid a \in L \setminus L_2 \text{ or } a \in L_2 \setminus L\}.$$

Expressing that solutions should be as close as possible to the original solution S, according to this criterion, is a simple matter. Here is the corresponding preference program:

$$P_{pref(S)} = \{a > not\ a \mid a \in S\} \cup \{not\ a > a \mid a \notin S\}.$$

Proposition 3. *Let S be a set of literals, P a program. S_1 is a preferred answer set of the optimization program $(P, P_{pref(S)})$ iff it is an answer set of P maximally close to S.*

To be more specific, assume we have to schedule 3 meetings m_1, m_2, m_3. Moreover, there are 4 available non-overlapping time slots $s_1, \ldots, s_4$ and 4 persons $p_1, \ldots, p_4$. We use $slot(M, S)$ to express that the time slot for meeting M is S, $part(P, M)$ to express that person P needs to participate in meeting M, and $unav(P, S)$ to express that person P is unavailable at time slot S. Assume we have

$$\begin{array}{lll} part(p_1, m_1) & part(p_3, m_2) & unav(p_1, s_4) \\ part(p_2, m_1) & part(p_3, m_3) & unav(p_2, s_4) \\ part(p_2, m_2) & part(p_4, m_3) & unav(p_4, s_2) \end{array}$$

Using cardinality constraints [27] we can express that each meeting needs exactly one time slot:

$$1\{slot(M, S) : slot(S)\}1 \leftarrow meeting(M)$$

Furthermore, we need the following constraints expressing that participants have to be available and that one cannot attend different meetings at the same time:

$$\leftarrow part(P, M), slot(M, S), unav(P, S)$$
$$\leftarrow part(P, M), part(P, M'), M \neq M', slot(M, S), slot(M', S)$$

One of the answer sets of this program contains:

$$slot(m_1, s_1), slot(m_2, s_2), slot(m_3, s_3)$$

Now assume p_4 becomes unavailable at s_3:

$$unav(p_4, s_3)$$

The former solution is no longer valid. Among the new solutions we have:

$$S_1 : slot(m_1, s_1), slot(m_2, s_2), slot(m_3, s_4)$$
$$S_2 : slot(m_1, s_2), slot(m_2, s_1), slot(m_3, s_4)$$
$$S_3 : slot(m_1, s_3), slot(m_2, s_2), slot(m_3, s_1)$$

Clearly, S_1 is closer to the original solution than S_2. Indeed, our criterion is satisfied and our optimality program will prefer S_1 over S_2.

But what about S_1 and S_3? This time the symmetric difference between S and S_1 and that between S and S_3 are not in subset relation: the former contains $slot(m_3, s_4)$ which is not in the latter, the latter contains $slot(m_1, s_3)$ which is not in the former. Intuitively, we may prefer S_1 since it changes fewer meetings in terms of cardinalities. This can simply be achieved by switching from an inclusion-based to a cardinality-based combination rule for satisfaction degrees. S_1 is better than S_2 since the number of preference rules satisfied to degree 2 in S_3 is higher than the number of preference rules satisfied to degree 2 in S_1.

The specification of the corresponding distance measure is straightforward and therefore omitted.

Numerous further refinements can be thought of: we may determine closeness based on the set (or number) of people participating in meetings whose slot is changed rather than on the set (or number) of meetings. We may also use meta preferences and give higher preference to rules minimizing changes for, say professors, followed by rules minimizing changes for assistants, followed by rules minimizing changes for students. All these refinements are simple exercises if optimization constructs are part of the ASP formalism, and become clumsy if they are not.

5 Inconsistency Handling

In classical logic there are two major ways of handling inconsistency: (1) one can weaken the set of premises, that is, disregard some of the premises entirely or replace them with weaker formulas such that the inconsistency disappears, or (2) one can weaken the inference relation to make sure that inconsistent premises will not lead to the derivation of arbitrary formulas. The former approach is often based on the notion of maximal consistent subsets of the premises, the latter uses paraconsistent logics which are often based on multi-valued logic.

In nonmonotonic reasoning and logic programming there is a third alternative: one can add information to the effect that a rule giving rise to inconsistency is blocked. A simple example is the program $\{p \leftarrow not\ p\}$ which can be made consistent[2] by adding the fact p.

An excellent overview of paraconsistent logic programming is [10]. We will stick here to the other approaches and show how optimization techniques can be used to make sure the effect of weakening or of adding new information is minimized. Let us start with a weakening approach.

Let P be a, possibly inconsistent, logic program. Let n be a function assigning unique names taken from a set N of atoms not appearing in P to each of the rules in P. We first replace P by the weakened program $weak_n(P)$:

$$\{head \leftarrow body, r \mid head \leftarrow body \in P, n(head \leftarrow body) = r\} \cup$$
$$\{r \leftarrow not\ \neg r \mid r \in N\} \cup$$
$$\{\neg r \leftarrow not\ r \mid r \in N\}$$

The new program indeed is very weak. Basically, the new representation allows us to switch rules on and off arbitrarily. Let $Lit(P)$ denote the set of literals built from atoms in P. We have the following result:

Proposition 4. *If S is an answer set of $weak_n(P)$ then $S \cap Lit(P)$ is an answer set of some subset of P. Vice versa, if S is an answer set of a subset of P, then there is an answer set S' of $weak_n(P)$ such that $S' \cap Lit(P) = S$.*

[2] We use the term inconsistent both for programs having an inconsistent answer set and for programs having no answer set at all. Some authors call the latter incoherent, but the distinction will not matter here.

Existence of a consistent answer set follows directly from this proposition. Of course, we do not want to consider arbitrary subsets of the original program but maximal ones. Using optimization programs, this can be achieved by using the set of preference rules:

$$\{r > \neg r \mid r \in N\}.$$

Preferred answer sets now correspond, up to literals built from atoms in N, to answer sets of maximal consistent subsets of the original program P.

Obviously this is still a very simple strategy. Optimization programs offer the possibility to express preferences among the rules to be taken into account. For instance, assume r_1 is a domain constraint and r_2 an observation made by a potentially unreliable observer. One would certainly consider r_1 as more entrenched (the terminology is taken form the area of belief revision) than r_2. We can model this simply by adding a statement of the form $r_1 > r_2$ to our preference program. We can also express context dependent preferences among rules. For instance, if we have meta information about the sources of rules, say $source(r_1, Peter)$, $source(r_2, John)$, together with information about the reliability of sources, say $more{-}reliable(Peter, John)$, then we can use a rule of the form

$$R_1 > R_2 \leftarrow source(R_1, S_1), source(R_2, S_2), more{-}reliable(S_1, S_2)$$

to give preference to rules with more reliable sources. Similarly, if we have temporal meta-information about when a rule was obtained, we may have rules expressing that information obtained earlier in time is less preferred than information obtained later. Preferences of this kind are common in the area of belief revision. In any case, given adequate meta information, optimization facilities allow us to represent declaratively our strategy of handling inconsistencies.

As mentioned in the beginning of this section consistency in logic programming can sometimes be restored by adding rather than disregarding rules.[3] Assume a set of rules R to be used for this purpose is given together with the original program P. Borrowing terminology from [23] we call them contradiction removal rules. Again we build the weakened program $weak_n(P \cup R)$ using a naming function n. We use N_P to denote the names of rules in P and N_R to denote the names of rules in R.

The major difference now lies in the preference program. Intuitively, we have to minimize the set of rules from P which are disregarded and to maximize the set of rules from R which are disregarded. It is a simple matter to express this using the preference program:

$$\{r > \neg r \mid r \in N_P\} \cup \{\neg r > r \mid r \in N_R\}.$$

Since it is the purpose of contradiction removal rules to be used whenever inconsistencies arise we may want to add meta preferences. If we give rules in the

[3] This approach is also pursued in CR-Prolog [3]. The method presented here differs from CR-Prolog in two respects: we can add as well as disregard rules, and preference programs allow us to describe flexible inconsistency handling strategies declaratively.

first of the two sets above preference over the rules in the second set, then the contradiction is removed by removal rules alone whenever this is possible. Only if the removal rules are insufficient, information from the original program is disregarded.

The reader will have observed that we used preference rules containing $>$ rather than LPOD rules containing $\times$ in this and the preceding section, whereas Sect. 3 was based on LPODs. This is not incidental. In Sect. 3 all preferences were about available options which needed to be generated anyway. The use of $\times$ is highly convenient in such cases. Here we need to be able to express, say, that a domain constraint r_1 is preferred over an observation r_2 *without* assuming that each answer set contains one of these two rules. The ability to express preferences without generating answer sets containing one of the options appears crucial here.

6 Requirements For A Qualitative Optimization Language

The examples in the last three sections have shown that qualitative optimization is an important issue for a variety of problems. In further work we are planning to develop a language for stating problems of this kind. From the discussion of our examples we derive the following requirements for such a language:

1. *modularity*: we want to be able to express optimization policies independently of the type of program used for answer set generation. This is a feature already realized in optimization programs and should remain a feature of any future extension of the underlying language.
2. *generality*: both ordered disjunction and preference statements based on $>$ as in optimization programs have their merits. The latter can express preferences based on formulas satisfied in answer sets, even if corresponding generating programs are not available. The former are highly convenient whenever preferences are expressed among options which have to be generated anyway, as it was the case in abduction and diagnosis. Therefore, both should be available in a general language.
   We also believe that a further generalization to partial orders will be useful. Although optimization programs allow us to rank classes of answer sets depending onthe boolean combinations they satisfy, these classes are still totally ordered by each preference rule. There will be relevant cases where a partial order is all we have.
3. *multiple combination strategies*: already the few examples discussed in this paper show that different strategies of combining preferences based on single rules to a global preference order on answer sets are necessary. We discussed in particular inclusion-based and cardinality-based combinations, but others may turn out to be useful as well.
   Since different strategies are useful the user should also have the possibility to combine different stategies in the same preference program. What we have

in mind are, possibly nested, blocks of preference rules such that each block has its own user defined combination strategy.

4. *meta preferences*: in the preceding sections we have already discussed several useful applications of meta-preferences among preference rules. For optimization programs meta-preferences and their semantics were defined in [6] based on a partition of the rules. If a general language for qualitative optimization has some kind of a block structure, as indicated in item 3. above, then the blocks are adequate structures to define meta preferences on. A partial preference order on - possibly nested - blocks of preference rules would be a useful generalization of the meta preferences we have right now.

7 Conclusions

In this paper we have demonstrated that optimization constructs beyond those available in current answer set solvers can be highly fruitful for numerous problem solving tasks. To support our claim we discussed several generic problems: abduction and diagnosis, solution coherence and inconsistency handling. Each of these problems may arise in a large number of specific applications, and for each of them optimization facilities lead to drastic simplifications of problem specifications. We therefore believe that combining current logic programming languages with a general language for qualitative optimization, for which we derived several requirements, will greatly increase the impact of ASP on AI and computer science in general.

Acknowledgements. Thanks to Hans Tompits and the anonymous reviewers for helpful comments.

References

1. C. Baral. *Knowledge representation, reasoning and declarative problem solving*, Cambridge University Press, 2003.
2. C. Baral, C. Uyan, Declarativ specification and solution of combinatorial auctions using logic programming, *Proc. LPNMR'01*, Springer Verlag, 2001.
3. M. Balduccini, M. Gelfond, Logic Programs with Consistency-Restoring Rules, In P. Doherty, J. McCarthy, M.-A. Williams (editors), *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, Mar 2003.
4. G. Brewka, Logic programming with ordered disjunction, In *Proc. AAAI-02*, 100–105, Morgan Kaufmann, 2002.
5. G. Brewka and T. Eiter, Preferred answer sets for extended logic programs, *Artificial Intelligence* 109: 297–356, 1999.
6. G. Brewka, I. Niemelä, and M. Truszczynski, Answer set optimization, *Proc. IJCAI-03*, 867–872, Acapulco, 2003.
7. G. Brewka, I. Niemelä, and T. Syrjänen, Implementing ordered disjunction using answer set solvers for normal programs, *Proc. JELIA 2002*, Springer Verlag, 2002.

8. F. Buccafurri, N. Leone, P. Rullo Enhancing disjunctive datalog by constraints, *IEEE Transactions on Knowledge and Data Engineering*, 12(5), 845–860, 2000.
9. L. Console, D.T. Dupre, P. Torasso, On the relatio between abduction and deduction, *Journal of Logic and Computation* 1(5):661–690, 1991.
10. C. V. Damasio and L. M. Pereira, A Survey on Paraconsistent Semantics for Extended Logic Programas, in: D.M. Gabbay and Ph. Smets (eds.), *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, vol. 2, pp 241–320, Kluwer Academic Publishers, 1998.
11. O. Dressler, P. Struss, The consistency-based approach to automated diagnosis of devices, in: G. Brewka (ed), *Principles of Knowledge Representation*, CSLI Publications, 1996.
12. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The kr system dlv: Progress report, comparisons and benchmarks, *Proc. Principles of Knowledge Representation and Reasoning, KR-98*, Morgan Kaufmann, 1998.
13. T. Eiter, W. Faber, N. Leone, G. Pfeifer, The Diagnosi Frontend of the dlv System, *AI Communications* 12(1-2): 99–111, 1999.
14. T. Eiter, M. Fink, G. Sabbatini, H. Tompits A generic approach for knowledge-based information-site selection, *Proc. Principles of Knowledge Representation and Reasoning*, KR-02, Toulouse, Morgan Kaufman, 2002
15. T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres, Answer se planning under action costs, *Proc. JELIA 2002*, Springer Verlag, 2002, extended version to appear in Journal of Artificial Intelligence Research.
16. K. Konolige, Abduction versus closure in causal theories, *Artificial Intelligence* 53: 255–272, 1992.
17. M. Gelfond and V. Lifschitz, Classical negation in logic programs and disjunctive databases, *New Generation Computing*, 9:365–385, 1991.
18. V. Lifschitz, Answer set programming and plan generation, *Artificial Intelligence*, 138(1-2):39–54, 2002.
19. V. Marek and M. Truszczynski, Stable models and an alternative logic programming paradigm, in: *The Logic Programming Paradigm: a 25-Year Perspective*, 1999.
20. I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
21. I. Niemelä and P. Simons, Efficient implementation of the stable model and well-founded semantics for normal logic programs, *Proc. 4th Intl. Conference on Logic Programming and Nonmonotonic Reasoning*, Springer Verlag, 1997.
22. Y. Peng and J. Reggia, Abductive inference models for diagnostic problem solving, *Symbolic Computation - Artificial Intelligence*, Springer, 1990.
23. L. M. Pereira, J. J. Alferes and J. Aparicio. Contradiction Removal within Well Founded Semantics, in: A. Nerode, W. Marek, V. S. Subrahmanian (editors), *Logic Programming and Nonmonotonic Reasoning*, pages 105–119, MIT Press, 1991.
24. D. Poole, An architecture for default and abductive reasoning, *Computational Intelligence*, 5(1): 97–110, 1989.
25. C. Sakama and K. Inoue, Prioritized logic programming and its application to commonsense reasoning, *Artificial Intelligence*, 123(1-2):185–222, 2000.
26. T. Schaub and K. Wang, A comparative study of logic programs with preference, In *Proc. Intl. Joint Conference on Artificial Intelligence, IJCAI-01*, 2001.
27. P. Simons, I. Niemelä, and T. Soininen, Extending and implementing the stable model semantics, *Artificial Intelligence*, 138(1-2):181–234, 2002.
28. T. Soininen, *An Approach to Knowledge Representation and Reasoning for Product Configuration Tasks*, PhD thesis, Helsinki University of Technology, Finland, 2000.

A Logic of Non-monotone Inductive Definitions and Its Modularity Properties

Marc Denecker[1] and Eugenia Ternovska[2]

[1] Department of Computer Science, K.U.Leuven, Belgium
marcd@cs.kuleuven.ac.be
[2] School of Computing Science, Simon Fraser University, Vancouver, Canada
ter@cs.sfu.ca

Abstract. Well-known principles of induction include monotone induction and different sorts of non-monotone induction such as inflationary induction, induction over well-ordered sets and iterated induction. In this work, we define a logic formalizing induction over well-ordered sets and monotone and iterated induction. Just as the principle of positive induction has been formalized in FO(LFP), and the principle of inflationary induction has been formalized in FO(IFP), this paper formalizes the principle of iterated induction in a new logic for Non-Monotone Inductive Definitions (NMID-logic). The semantics of the logic is strongly influenced by the well-founded semantics of logic programming.
Our main result concerns the modularity properties of inductive definitions in NMID-logic. Specifically, we formulate conditions under which a simultaneous definition Δ of several relations is logically equivalent to a conjunction of smaller definitions $\Delta_1 \wedge \cdots \wedge \Delta_n$ with disjoint sets of defined predicates. The difficulty of the result comes from the fact that predicates P_i and P_j defined in Δ_i and Δ_j, respectively, may be mutually connected by simultaneous induction. Since logic programming and abductive logic programming under well-founded semantics are proper fragments of our logic, our modularity results are applicable there as well. As an example of application of our logic and theorems presented in this paper, we describe a temporal formalism, the inductive situation calculus, where causal dependencies are naturally represented as rules of inductive definitions.

1 Introduction

This paper fits into a broad project aiming at studying general forms of inductive definitions and their role in diverse fields of mathematics and computer science. Monotone inductive definitions and inductive definability have been studied extensively in mathematical logic [Mos74,Acz77]. The algebraic foundations for monotone induction are laid by Tarski's fixed point theory of monotone lattice operators [Tar55]. The notion of an inductive definition is the underlying concept in fixed point logics [DG02], which are closely connected to database query languages. Fixed point constructs for monotone inductive and co-inductive definitions play a central role in the specification languages for formal verification

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 47–60, 2004.

of dynamic systems such as the μ-calculus. Induction axioms have been used successfully in proving properties of protocols using automated reasoning tools, and the concept of a definition is fundamental in the area of description logics. Thus, it appears that the notion of a definition and its inductive generalizations emerges as a unifying theme in many areas of mathematics and computational science. Hence, the study of inductive definitions could provide further insight in the interrelations between these areas and lead to synergy between them.

Despite an abundance of logics that capture some aspects of inductive definitions, there is no sufficiently general logic that represents monotone induction, and, at the same time, captures non-monotone forms of induction, such as iterated induction and induction over well-ordered sets. Recently, [Den98,DBM01] have argued that semantical studies of logic programming may contribute to a better understanding of such generalized forms of induction. In particular, it was argued that the well-founded semantics of logic programming [VRS91] formalizes and generalizes the non-monotone forms of induction mentioned above. In [DMT00], a fixed point theory of general non-monotone lattice operators, called *approximation theory*, was developed. This theory generalizes Tarski's theory of fixed points of monotone lattice operators, and provides the algebraic foundation of this generalized principle of iterated induction modeled by the well-founded semantics.

Our main objective here is to study modularity properties of non-monotone induction. Modularity is widely recognized as an important property in formal verification and knowledge representation. For example, in the context of specification and verification of complex dynamic systems, it is crucial that we can specify a complex system by describing its components in independent modules which can then be conjoined to form a correct description of the complete system. Thus, the operation of joining modules should preserve the correctness of the component modules. The dual operation of *splitting* a complex theory into an equivalent set of smaller modules is equally important. It allows to investigate complex theories by studying its modules independently, and reduces the analysis of the correctness of the complex theory to the much simpler problem of analyzing the correctness of its modules.

Our main result is a set of techniques that allow one to break up a definition into a conjunction of smaller and simpler definitions. More precisely, we investigate general formal conditions that guarantee that a simultaneous definition of several predicates can be split into the conjunction of a set of components of this definition, each component defining a subset of the defined predicates. Our theorems also provide conditions under which joining a set of definitions for distinct sets of predicates into one simultaneous definition is equivalence preserving. The problem we study is similar to that studied in [Sch95,VDD00], but our results are uniformly stronger in that they are proven for a more expressive logic and under more general conditions.

The paper is structured as follows. In section 2, we extend classical logic with generalized non-monotone definitions. In section 3, the modularity is investigated. Then we demonstrate an application of our logic on the example of

a non-monotonic reasoning formalism of the inductive situation calculus. We conclude with section 5, where we summarize our results, and outline future directions.

2 NMID-Logic

In this section, we present an extension of classical logic with non-monotone inductive definitions. This work extends previous work of [Den00].

Syntax First, we introduce the notion of a definition. We introduce a new binary connective $\leftarrow$, called the *definitional implication*. A *definition* Δ is a set of rules of the form

$$\forall \bar{x}\ (X(\bar{t}) \leftarrow \varphi) \quad \text{where} \tag{1}$$

- $\bar{x}$ is a tuple of object variables,
- X is a predicate symbol (i.e., a predicate constant or variable) of some arity r,
- $\bar{t}$ is a tuple of terms of length r of the vocabulary τ,
- φ is an arbitrary first-order formula of τ.

The definitional implication $\leftarrow$ must be distinguished from material implication. A rule $\forall \bar{x}\ (X(\bar{t}) \leftarrow \varphi)$ in a definition does not correspond to the disjunction $\forall \bar{x}(X(\bar{t}) \vee \neg\varphi)$. Note that in front of rules, we allow only universal quantifiers. In the rule (1), $X(\bar{t})$ is called the *head* and φ is the *body* of the rule.

Example 1. The following expression is a simultaneous definition of even and odd numbers on the structure of the natural numbers with zero and the successor function: $\{\forall x(E(x) \leftarrow x = 0), \forall x(E(s(x)) \leftarrow O(x)), \forall x(O(s(x)) \leftarrow E(x))\}$. An example of a non-monotone definition in the well-ordered set of natural numbers is the expression $\{\forall x(E(x) \leftarrow x = 0), \forall x(E(s(x)) \leftarrow \neg E(x))\}$.

A *defined symbol* of Δ is a relation symbol that occurs in the head of at least one rule of Δ; other relation, object and function symbols are called *open*. In the Example 1 above, E and O are defined predicate symbols, and s is an open function symbol.

Remark 1. In most parts of this paper, we do not make a formal distinction between variable and constant symbols. Symbols occurring free in a formula can be viewed as constants; symbols in the scope of a quantifier can be viewed as variables. In examples, we tend to quantify over x, y, X, Y, and leave c, g, f and P, Q free and treat them as constants.

Let $free(\phi)$ denotes the set of free symbols of ϕ. We shall use notation $\phi(x_1, \ldots x_n)$ to emphasize that symbols $x_1, \ldots x_n$ are distinct and are free in ϕ. Let τ be a vocabulary interpreting all free symbols of Δ. The subset of defined symbols of definition Δ is denoted $\tau^{\mathrm{d}}_{\Delta}$. The set of open symbols of Δ in τ is denoted $\tau^{\mathrm{o}}_{\Delta}$. The sets $\tau^{\mathrm{d}}_{\Delta}$ and $\tau^{\mathrm{o}}_{\Delta}$ form a partition of τ, i.e., $\tau^{\mathrm{d}}_{\Delta} \cup \tau^{\mathrm{o}}_{\Delta} = \tau$, and $\tau^{\mathrm{d}}_{\Delta} \cap \tau^{\mathrm{o}}_{\Delta} = \emptyset$. A *well-formed formula* of the Logic for Non-Monotone Inductive Definitions, briefly a NMID-formula, is defined by the following induction:

- If X is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms then $X(t_1, \ldots, t_n)$ is a formula.
- If Δ is a definition then Δ is a formula.
- If ϕ, ψ are formulas, then so are $(\neg\phi)$ and $(\phi \wedge \psi)$.
- If ϕ is a formula, then $\exists\sigma\ \phi$ is a formula.

A formula ϕ is an NMID-formula over a vocabulary τ if $free(\phi) \subseteq \tau$. We use SO(NMID)$[\tau]$ to denote the set of all formulas of our logic over fixed vocabulary τ. The first order fragment FO(NMID)$[\tau]$ is defined in the same way, except that quantification over set and function symbols is not allowed.

Example 2. In the language of the natural numbes, the following formula expresses that there is a set which is the least set containing 0 and closed under taking successor numbers, and which contains all domain elements. It is equivalent with the standard induction axiom:

$$\exists N[\{\forall x(N(x) \leftarrow x = 0), \forall x(N(s(x)) \leftarrow N(x))\} \wedge \forall x\ N(x)].$$

Semantics. The exposition below is a synthesis of different approaches to the well-founded semantics, in particular those presented in [Van93,Fit03,DBM01]. We assume familiarity with semantics of classical logic. A structure I of a vocabulary τ consists of a domain $dom(I)$ and for each symbol $\sigma \in \tau$ a *value* σ^I in $dom(I)$, i.e. a function for a function symbol and a relation for a predicate symbol of corresponding arity. The value t^I of a term t in I is defined by the standard recursion. The structure obtained from I by assigning value v to symbol σ is denoted $I[\sigma : v]$.

Let Δ be an arbitrary definition. For each defined symbol X of Δ, we define:

$$\varphi_X(\bar{x}) := \exists\bar{y}_1\ \bar{x} = \bar{t}_1 \wedge \varphi_1 \vee\ \cdots \vee\ \exists\bar{y}_m\ \bar{x} = \bar{t}_m \wedge \varphi_m, \tag{2}$$

where $\bar{x}$ is a tuple of new variables, and $\forall\bar{y}_1\ (X(\bar{t}_1) \leftarrow \varphi_1), \ldots, \forall\bar{y}_m\ (X(\bar{t}_m) \leftarrow \varphi_m)$ are the rules of Δ with X in the head.

The basis of the construction of the well-founded model is an operator T_Δ mapping pairs of τ-structures to τ-structures. Given such a pair (I, J), the operator T_Δ evaluates the bodies of the rules in a particular way. In particular, it evaluates positive occurrences of defined symbols in rule bodies by I, and negative occurrences of defined symbols by J. To formally define this operator, we simply rename the negative occurrences in rule bodies of Δ. We extend the vocabulary τ with, for each defined symbol X, a new relation symbol X' of the same arity. The extended vocabulary $\tau \cup \bar{X}'$ will be denoted τ'. Then in each rule body in Δ, we substitute the symbol X' for each negative occurrence of a defined symbol X, thus obtaining a new definition Δ'.
Now all defined predicates of Δ' occur positively in the bodies of the rules, so Δ' is a positive definition based on vocabulary τ'. The new definition Δ' has the same defined symbols as Δ, and the symbols with primes are its new open symbols.

Let Δ' be obtained from Δ as described. As in the case of Γ_Δ, we obtain formula (2), but now we use Δ' instead of Δ. For any pair of τ-structures I, J

which share the same domain, define I_J as the τ'-structure which satisfies the following:

- its domain is the same as the domain of I and J
- each open symbol of Δ is interpreted by J
- each defined symbol of Δ is interpreted by I
- the value of each new symbol X' is X^J, the value of X in J.

It is clear that for some defined symbol X, evaluating φ'_X under I_J simulates the non-standard structure of φ_X where J is "responsible" for the open and the negative occurrences of the defined predicates, while I is "responsible" for the positive ones.

Definition 1 (operator T_Δ). *We introduce a partially defined binary operator $\mathsf{T}_\Delta : \mathcal{I} \times \mathcal{I} \mapsto \mathcal{I}$, where $\mathcal{I}$ is the class of all τ-structures. The operator is defined on pairs of structures which share the same domain, and is undefined otherwise. We have $I' = \mathsf{T}_\Delta(I, J)$ if and only if*

- *$dom(I') = dom(J) = dom(I)$,*
- *for each open symbol σ, $\sigma^{I'} := \sigma^J$ and*
- *for each defined symbol $X \in \tau^{\mathrm{d}}_\Delta$, $X^{I'} := \{\bar{a} \mid I_J \models \varphi'_{X_i}[\bar{a}]\}$ where formula φ'_X is defined by equation (2) applied to Δ'.*

Let J be a τ-structure, and $I_{\mathrm{o}} := J|_{\tau^{\mathrm{o}}_\Delta}$, the restriction of J to τ^{o}_Δ. The unary operator $\lambda J\ \mathsf{T}_\Delta(I, J)$, often denoted by $\mathsf{T}_\Delta(\cdot, J)$, is a monotone operator; and its least fixpoint in this lattice is computed by $\mathit{lfp}(\mathsf{T}_\Delta(\cdot, J)) := \bigsqcup_\xi E^\xi$, where $E^\xi := \mathsf{T}_\Delta(E^{<\xi}, J)$, and $E^{<\xi} := \bigsqcup_{\eta<\xi} E^\eta$.

Definition 2 (stable operator). *Define the* stable[1] operator $ST_\Delta : \mathcal{I} \mapsto \mathcal{I}$ *as follows:* $ST_\Delta(J) := \mathit{lfp}(\mathsf{T}_\Delta(\cdot, J))$.

Operator ST_Δ is anti-monotone. As is standard for anti-monotone operators, the operator ST_Δ gives rise to a sequence $(I^\xi, J^\xi)_{\xi\geq 0}$ defined by

$$\begin{array}{l} I^\xi := ST_\Delta(J^{<\xi}), \quad \text{where} \quad J^{<\xi} := \sqcap_{\eta<\xi} J^\eta, \\ J^\xi := ST_\Delta(I^{<\xi}), \quad \text{where} \quad I^{<\xi} := \sqcup_{\eta<\xi} I^\eta. \end{array}$$

Notice that $I^{<0}$ is, by definition, the bottom element $\bot_{I_{\mathrm{o}}}$ of the lattice of τ-structures extending I_{o}, i.e., the structure which assigns $\emptyset$ to every defined symbol; and $J^{<0}$ is the top element $\top_{I_{\mathrm{o}}}$ which assigns Cartesian product A^r to each r-ary defined symbol X of Δ.

The anti-monotonicity of ST_Δ implies that the sequence $(I^\xi)_{\xi\geq 0}$ is increasing and $(J^\xi)_{\xi\geq 0}$ is decreasing. Moreover, for each ξ, $I^\xi \sqsubseteq J^\xi$. Thus, it holds that the sequence $(I^\xi, J^\xi)_{\xi\geq 0}$ is indeed a sequence of increasingly precise approximations. This sequence has a limit (I, J), which is the maximal oscillating pair of ST_Δ.

[1] This operator is often called the Gelfond-Lifschitz operator. The corresponding transformation was introduced in [GL91].

Equivalently, I and J are fixpoints of the square ST^2_Δ, $lfp(ST^2_\Delta)$ and $gfp(ST^2_\Delta)$, respectively.

We define ${I_o}^{\Delta\downarrow} := lfp(ST^2_\Delta)$, and ${I_o}^{\Delta\uparrow} := gfp(ST^2_\Delta)$. We extend this notation to any structure K which interprets at least τ^o_Δ and define $K^{\Delta\downarrow} := (K|_{\tau^o_\Delta})^{\Delta\downarrow}$, and $K^{\Delta\uparrow} := (K|_{\tau^o_\Delta})^{\Delta\uparrow}$. Note that $K^{\Delta\downarrow}$ and $K^{\Delta\uparrow}$ agree with K on the open symbols but not necessarily on the defined symbols.

Definition 3. *Definition Δ is* total *in τ^o_Δ-structure I_o if ${I_o}^{\Delta\downarrow} = {I_o}^{\Delta\uparrow}$.*

The aim of an inductive definition is to *define* its defined symbols. Therefore, a natural quality requirement for a definition is that it is total. We will define that τ-structure I satisfies Δ, or equivalently, that Δ is true in I (denoted $I \models \Delta$) if Δ is total in I and $I^{\Delta\downarrow}$ is identical to I.

Definition 4 (Δ-extension of I). *Let Δ be a definition. Let I be a structure of vocabulary τ_I such that $\tau^o_\Delta \subseteq \tau_I \subseteq \tau$. If Δ is total in I, define the* Δ-extension of I, *denoted I^Δ, as $I^\Delta := I^{\Delta\downarrow}$ (or, equivalently, $I^\Delta := I^{\Delta\uparrow}$). If Δ is not total in I, then I has no Δ-extension.*

Note that for any τ^o_Δ-structure I_o, there is at most one Δ-extension extending I_o.

Definition 5 (ϕ true in structure I). *Let ϕ be a NMID-formula and I any structure such that $free(\phi) \subseteq \tau_I$. We define $I \models \phi$ (in words, ϕ* is true in *I, or I* satisfies *ϕ, or I is a* model *of ϕ) by the following induction:*

- *$I \models X(t_1, .., t_n)$ if $(t^I_1, .., t^I_n) \in X^I$;*
- *$I \models \psi_1 \wedge \psi_2$ if $I \models \psi_1$ and $I \models \psi_2$;*
- *$I \models \neg\psi$ if $I \not\models \psi$;*
- *$I \models \exists\sigma\ \psi$ if for some value v of σ in the domain $dom(I)$ of I, $I[\sigma : v] \models \psi$;*
- *$I \models \Delta$ if $I = I^{\Delta\downarrow} = I^{\Delta\uparrow}$.*

Given an NMID-theory T over τ, a τ-structure I satisfies T (is a model of T) if I satisfies each $\phi \in T$. This is denoted by $I \models T$.

3 Modularity

In this section, we split a definition Δ into subdefinitions $\{\Delta_1, \Delta_2, \ldots, \Delta_n\}$. We study under what conditions we can guarantee that for structure I, $I \models \Delta$ iff $I \models \Delta_1 \wedge \Delta_2 \wedge \cdots \wedge \Delta_n$. This is the subject of the Modularity theorem.

Definition 6 (partition of definitions). *A* partition of definition *Δ is a set $\{\Delta_1, \ldots, \Delta_n\}$, $1 < n$, such that $\Delta = \Delta_1 \cup \cdots \cup \Delta_n$, and if defined symbol P appears in the head of a rule of Δ_i, $1 \leq i \leq n$, then all rules of Δ with P in the head belong to Δ_i.*

Notice that Δ_i has some "new" open symbols. For instance, if P is defined in Δ, but not in Δ_i, then it is a new open symbol of Δ_i. Of course, it holds that $\tau = \tau^{\mathrm{o}}_{\Delta} \cup \tau^{\mathrm{d}}_{\Delta} = \tau^{\mathrm{o}}_{\Delta_i} \cup \tau^{\mathrm{d}}_{\Delta_i}$, $1 \leq i \leq n$. Also, $\cup_i \tau^{\mathrm{d}}_{\Delta_i} = \tau^{\mathrm{d}}_{\Delta}$ and $\tau^{\mathrm{d}}_{\Delta_i} \cap \tau^{\mathrm{d}}_{\Delta_j} = \emptyset$ whenever $i \neq j$.

The following theorem demonstrates that a model of a definition, is, at the same time, a model of each its sub-definitions. As a side effect, we demonstrate that the totality of the large definition implies the totality of its sub-definitions.

Theorem 1 (decomposition). *Let Δ be a definition over τ with partition $(\Delta_1, \ldots, \Delta_n)$. Let I be a τ-structure. Then $I \models \Delta$ implies $I \models \Delta_1 \wedge \ldots \wedge \Delta_n$.*

Let I be a model of Δ. The theorem says that for each i, $1 \leq i \leq n$, Δ_i is total in the restriction $I|_{\tau^{\mathrm{o}}_{\Delta_i}}$ of I to the open symbols of Δ_i and moreover that I is the Δ_i-extension of $I|_{\tau^{\mathrm{o}}_{\Delta_i}}$. Using the notations of Definition 4, this means that $I = I^{\Delta_i \downarrow} = I^{\Delta_i \uparrow}$. We obtain the following corollary.

Corollary 1. *If $I \models \Delta$, then for each i, $1 \leq i \leq n$, Δ_i is total in $I|_{\tau^{\mathrm{o}}_{\Delta_i}}$.*

The following example shows that the inverse direction of theorem 1 does not hold in general.

Example 3. Let Δ, Δ_1, Δ_2 be the following definitions.

$$\Delta := \left\{ \begin{array}{l} P \leftarrow Q, \\ Q \leftarrow P \end{array} \right\} \quad \Delta_1 := \left\{ P \leftarrow Q \right\} \quad \Delta_2 := \left\{ Q \leftarrow P \right\}.$$

Definition Δ is total, and its unique model is $\emptyset$ in which both P and Q are false. According to theorem 1, $\emptyset$ satisfies Δ_1 and Δ_2. Note that $\{P, Q\}$ is not a model of Δ and yet, it satisfies Δ_1 and Δ_2. Indeed, $\{P, Q\}$ is the Δ_1-extension of the $\tau^{\mathrm{o}}_{\Delta_1}$-structure $\{Q\}$ and the Δ_2-extension of the $\tau^{\mathrm{o}}_{\Delta_2}$-structure $\{P\}$. The reason why the equivalence between $\Delta_1 \wedge \Delta_2$ and Δ breaks down is that in Δ there is a circular dependency between P and Q. After splitting, this circular dependency is broken up into two non-circular dependencies.

The example suggests that splitting a definition will be equivalence preserving if the splitting does not break circular dependencies between atoms. To this end, we introduce the notion of a *reduction relation*. A reduction relation will be defined as a binary relation on ground atoms, which expresses possible dependencies between atoms in the definition.

A domain atom over vocabulary τ in domain A is any atom $P(x_1, \ldots, x_n)$ where P is relation symbol of τ and $x_1, \ldots, x_n$ are elements of A. Let At^{τ}_{A} be the set of domain atoms over τ in A. Let $\prec$ be any binary relation on At^{τ}_{A}. If $Q[\bar{b}] \prec P[\bar{a}]$, we will say that $P[\bar{a}]$ depends on $Q[\bar{b}]$ (according to $\prec$). We use $Q[\bar{b}] \prec\!\!\!\prec P[\bar{a}]$ as an abbreviation for $Q[\bar{b}] \prec P[\bar{a}] \wedge P[\bar{a}] \not\prec Q[\bar{b}]$. For any domain atom $P[\bar{a}]$ and any pair I, J of τ-structures with domain A, we define $I \cong_{\prec P[\bar{a}]} J$ if for each atom $Q[\bar{b}] \prec P[\bar{a}]$, $I \models Q[\bar{b}]$ iff $J \models Q[\bar{b}]$. We extend this to pairs by defining $(I, J) \cong_{\prec P[\bar{a}]} (I', J')$ if $I \cong_{\prec P[\bar{a}]} I'$ and $J \cong_{\prec P[\bar{a}]} J'$.

Let K_o be a structure with domain A interpreting at least all object and function symbols of τ and no defined predicates of Δ.

Definition 7 (reduction relation). *A binary relation $\prec$ on At^τ_A is a* reduction relation *(or briefly, a reduction) of Δ in K_o if for each domain atom $P[\bar{a}]$ with P a defined symbol, for all τ-structures I, J, I', J' extending K_o, if $(I, J) \cong_{\prec P[\bar{a}]} (I', J')$ then $I_J \models \varphi_P[\bar{a}]$ iff $I'_{J'} \models \varphi_P[\bar{a}]$.*

Intuitively, the definition expresses that $\prec$ is a reduction relation if the truth of the formulas $\varphi_P[\bar{a}]$ depends only on the truth of the atoms on which $P[\bar{a}]$ depends according to $\prec$.

Recall that a pre-well-order is a reflexive and transitive relation such that every non-empty subset contains a minimal element. Note that every definition Δ has a trivial reduction pre-well-order in every τ^o_Δ-structure. Indeed, the total binary relation on At^τ_A trivially satisfies the conditions of a reduction pre-well-order. The following definition is crucial for the right-to-left direction of the Modularity theorem.

Definition 8 (reduction partition). *Call partition $\{\Delta_1, \ldots, \Delta_n\}$ of definition Δ a* reduction partition *of Δ in τ^o_Δ-structure I_o if there is a reduction pre-well-order $\prec$ of Δ in I_o and if $Q[\bar{b}] \prec P[\bar{a}]$ and $P[\bar{a}]$, $Q[\bar{b}]$ are not defined in the same Δ_i then $Q[\bar{b}] \not\approx P[\bar{a}]$.*

The intuition underlying this definition is that in a reduction partition, if an atom defined in one module depends on an atom defined in another module, then the latter atom is strictly less in the reduction ordering and hence does not depend on the first atom.

A partition $\{\Delta_1, \ldots, \Delta_n\}$ of Δ is called total in K_o if each Δ_i is total in K_o.

Theorem 2 (modularity). *If $\{\Delta_1, \ldots, \Delta_n\}$ is a total reduction partition of Δ in τ^o_Δ-structure I_o, then for any τ-structure M extending I_o, $M \models \Delta_1 \wedge \cdots \wedge \Delta_n$ iff $M \models \Delta$.*

Corollary 2. *Let T_o a theory over τ^o_Δ such that for any τ^o_Δ-model M_o of T_o, $\{\Delta_1, \ldots, \Delta_n\}$ is a total reduction partition of Δ in M_o.*

Then $T_o \wedge \Delta$ and $T_o \wedge \Delta_1 \wedge \cdots \wedge \Delta_n$ are logically equivalent.

Now we consider for two special cases of definitions how to translate them in classical logic. Let Δ be a positive definition, i.e. with only positive occurrences of defined symbols in rule bodies, defining the symbols $\bar{P}$. Let X_i and P_i have the same arity. Define $PID(\Delta) := \bigwedge \Delta \wedge \forall \bar{X}(\bigwedge \Delta[\bar{P}/\bar{X}] \rightarrow \bar{P} \subseteq \bar{X})$. Here, $\bigwedge \Delta$ is the conjunction of formulas obtained by replacing definitional with material implications in Δ, $\Delta[\bar{P}/\bar{X}]$ is the definition obtained by substituting X_i for each defined symbol P_i and $\bar{P} \subseteq \bar{X}$ is a shorthand for the formula $(\forall \bar{x} P_1(\bar{x}) \rightarrow X_1(\bar{x})) \wedge \cdots \wedge (\forall \bar{x} P_n(\bar{x}) \rightarrow X_n(\bar{x}))$. The formula $PID(\Delta)$ is the standard second-order formula to express that predicates $\bar{P}$ satisfy the positive inductive definition Δ.

Theorem 3. *Δ is total in each τ^o_Δ-structure. For any structure I, $I \models \Delta$ iff $I \models PID(\Delta)$.*

Define the *completion of* Δ, denoted $comp(\Delta)$, as the conjunction, for each defined symbol X of Δ, of formulas $\forall \bar{x}(X(\bar{x}) \leftrightarrow \varphi_X[\bar{x}])$. A reduction relation $\prec$ of Δ on At_A^τ is *strict* in K_o if it is a strict well-founded order (i.e., an antisymmetric, transitive binary relation without infinite descending chains). Thus, if $P \prec Q$ holds, then the bodies of rules defining Q may depend on the truth value of P, but not vice versa.

Theorem 4 (completion). *Suppose* $\prec$ *is a strict reduction relation of* Δ *in* τ_Δ^o*-structure* I_o*. Then (a) definition* Δ *is total in* I_o*, and (b) the model* ${I_o}^\Delta$ *of* Δ *is the unique model of* $comp(\Delta)$ *extending* I_o*.*

4 Example: Inductive Situation Calculus

The vocabulary τ_{sc} of the situation calculus (see, e.g. [Rei01]) is a many-sorted vocabulary with equality with sorts for actions (*Act*), situations (*Sit*), and possibly a finite number of domain specific sorts called object sorts ($Ob_1, \ldots, Ob_k$). The vocabulary contains a potentially infinite set of domain-dependent function symbols of the sort *Act*. Such functions take only objects of object sorts as their arguments. For example, in the block world domain, we may have actions $pick\_up(x)$ of arity 1 and $put\_on(x, y)$ of arity 2. The constant S_0 of sort *Sit* denotes the initial situation. Function *do* of sort *Sit* maps actions and situations to situations, i.e., given a and s, term $do(a, s)$ denotes the successor situation which is obtained from situation s by performing action a. The predicate constants $F_1, F_2, \ldots$ are called *fluents* and denote properties of the world (both in the initial situation and in other situations). They always have exactly one argument of sort *Sit* while the sort of each other argument is an object sort. For example $On(x, y, s)$ of arity 3 denotes that object x is on object y in situation s.

Here we define a variant of Reiter-style situation calculus, which we call the inductive situation calculus. All fluents will be defined by simultaneous induction on the well-ordered set of situations. Ramifications describing propagation of effects of actions are modeled as monotone inductions at the level of situations. The result is an iterated inductive definition with alternating phases of monotone and non-monotone induction. Below we describe the components of the inductive situation calculus.

The *foundational axioms of the inductive situation calculus*, $\mathcal{D}_f$, are the set of axioms consisting of the unique name axioms for situations

$$\begin{array}{c} \forall a\ \forall s\ \neg(S_0 = do(a, s)), \\ \forall a_1 \forall a_2 \forall s_1 \forall s_2\ (do(a_1, s_1) = do(a_2, s_2) \rightarrow a_1 = a_2 \wedge s_1 = s_2), \end{array} \tag{3}$$

and the domain closure axiom for situations

$$\exists X \left\{ \begin{array}{l} \forall s\ (X(s) \leftarrow s = S_0), \\ \forall s'\ \forall a\ (X(s') \leftarrow X(do(a, s'))) \end{array} \right\} \wedge \forall s\ X(s)). \tag{4}$$

The role of axiom (4) is to guarantee that the domain of situations *Sit* is the smallest set closed under applications of the function symbol *do*, which satisfies

the unique name axioms for situations. It is equivalent to Reiter's induction axiom for situations.

The second and main part of the inductive situation calculus is the definition Δ_{sc} defining the fluents. To define it, the vocabulary τ_{ind} of the inductive situation calculus extends τ_{sc} by three types of symbols. For each fluent F, the symbol I_F is used to describe the fluent in the initial situation and has the same object arguments as F but misses the situation argument of F. The other two types of symbols are symbols denoting causality predicates. For each fluent F, we introduce predicates C_F and $C_{\neg F}$ with the same arguments as F plus one argument of sort *Act*. The atom $C_F(\bar{x}, a, s)$ means that action a has the direct or indirect effect of initiating F for objects $\bar{x}$ in situation s. Likewise, $C_{\neg F}(\bar{x}, a, s)$ represents that a has the effect of terminating F for $\bar{x}$ in situation s.

The definition Δ_{sc} simultaneously defines all fluents and causality predicates in terms of the initial situation predicates. This definition can be split in the partition $\{\Delta_{\text{fluent}}, \Delta_{\text{effect}}\}$. The subdefinition Δ_{fluent} consists of for each fluent F, the following three rules

$$\begin{aligned}
&\forall a \forall s \forall \bar{x}\ (F(\bar{x}, S_0) \leftarrow I_F(\bar{x})),\\
&\forall a \forall s \forall \bar{x}\ (F(\bar{x}, do(a, s)) \leftarrow C_F(\bar{x}, a, s)),\\
&\forall a \forall s \forall \bar{x}\ (F(\bar{x}, do(a, s)) \leftarrow F(\bar{x}, s) \wedge \neg C_{\neg F}(\bar{x}, a, s))
\end{aligned}$$

The first rule of this subdefinition simply associates the fluent in the initial situation with its initial situation predicate. The second rule says that if a fluent is caused to hold in a situation, then it holds in that situation. The third rule models the inertia law.

The set Δ_{effect} defines all the causality predicates and contains rules of the form:

$$\begin{aligned}
&\forall a \forall s \forall \bar{x}\ (C_F(\bar{x}, a, s) \leftarrow \gamma_F^+(\bar{x}, a, s))\\
&\forall a \forall s \forall \bar{x}\ (C_{\neg F}(\bar{x}, a, s) \leftarrow \gamma_F^-(\bar{x}, a, s))
\end{aligned}$$

where $\gamma_F^+(\bar{x}, a, s)$ and $\gamma^- F(\bar{x}, a, s)$ are formulas with only positive occurrences of causality predicates and in which the free variable s is the situation argument of all fluent and causality predicates. Notice that the definitional implications, $\leftarrow$, in these definitions represent causal dependencies. The semantic correspondence between causality rules and rules in an inductive definition was first pointed out in [DTV98].

A third and last part of the inductive situation calculus is the *description of the initial situation*, $\mathcal{D}_{S_0}$. It is a domain dependent theory containing neither fluent nor causality symbols. For example, in case of complete knowledge of the initial situations, $\mathcal{D}_{S_0}$ may consist of a set of definitions of the initial predicates I_F.

An inductive situation calculus consists then of the elements $\mathcal{D}_{\text{f}} \cup \{\Delta_{sc}\} \cup \mathcal{D}_{S_0}$.

Example 4 (gear wheels). Let us describe a simple idealized mechanical system of two connected gear wheels. Suppose fluents T_1, T_2 represent the fact that the first (respectively, the second) wheel is turning. If a wheel starts or stops turning, we denote this change by action $Start_1$, $Start_2$, $Stop_1$, or $Stop_2$, respectively.

The following definitions specify direct and indirect effects of actions:

$$\Delta^{\text{wheels}}_{\text{effect}} := \left\{ \begin{array}{ll} \forall a \forall s\ (C_{T_1}(a,s) \leftarrow a = Start_1), & \forall a \forall s\ (C_{T_2}(a,s) \leftarrow a = Start_2), \\ \forall a \forall s\ (C_{\neg T_1}(a,s) \leftarrow a = Stop_1), & \forall a \forall s\ (C_{\neg T_2}(a,s) \leftarrow a = Stop_2) \\ \forall a \forall s\ (C_{T_1}(a,s) \leftarrow C_{T_2}(a,s)), & \forall a \forall s\ (C_{T_2}(a,s) \leftarrow C_{T_1}(a,s)) \\ \forall a \forall s\ (C_{\neg T_1}(a,s) \leftarrow C_{\neg T_2}(a,s)), & \forall a \forall s\ (C_{\neg T_2}(a,s) \leftarrow C_{\neg T_1}(a,s)) \end{array} \right\}$$

Each rule represents a local property of the system, namely an individual causal effect. By combining them we obtain a modular description of the entire system. Notice that the causal dependencies form a positive cycle.

Define $\Delta^{\text{wheels}}_{\text{fluent}}$ as the set of rules defining fluents T_1 and T_2 and $\Delta^{\text{wheels}}_{\text{sc}} := \Delta^{\text{wheels}}_{\text{fluent}} \cup \Delta^{\text{wheels}}_{\text{effect}}$. Let in the initial situation neither wheel be turning:

$$\mathcal{D}^{\text{wheels}}_{\text{init}} := \{\neg I_{T_1} \wedge \neg I_{T_2}\}.$$

Finally, let $\mathcal{D}^{\text{wheels}}_{\text{una(Act)}}$ be the unique name axioms for the sort *Act*. Thus, the gear wheels example is axiomatized by

$$\mathcal{D}^{\text{wheels}} := \mathcal{D}_{\text{f}} \cup \{\Delta^{\text{wheels}}_{sc}\} \cup \mathcal{D}^{\text{wheels}}_{\text{init}} \cup \mathcal{D}^{\text{wheels}}_{\text{una(Act)}}.$$

Our goal now is to simplify our axiomatization of this example. Using the properties proven in the previous sections, we are able to simplify our axiomatization of the Gear Wheels example by reducing definitions to formulas in classical logic.

Proposition 1. *The theory* $\mathcal{D}^{\text{wheels}}$ *is logically equivalent to*

$$\mathcal{D}_{\text{f}} \cup comp(\mathcal{D}^{\text{wheels}}_{\text{fluent}}) \cup PID(\Delta^{\text{wheels}}_{\text{effect}}) \cup \mathcal{D}^{\text{wheels}}_{\text{init}} \cup \mathcal{D}^{\text{wheels}}_{\text{una(Act)}}.$$

Proof. We give a sketch of the proof to illustrate how the theorems of Section 3 can be used.

First, we show that $\Delta^{\text{wheels}}_{\text{fluent}}$ is total in each structure satisfying the foundational axioms $\mathcal{D}_{\text{f}}$ and moreover, that

$$\mathcal{D}_{\text{f}} \models \Delta^{\text{wheels}}_{\text{fluent}} \leftrightarrow comp(\Delta_{\text{fluent}}).$$

To this aim, we show that in each structure I satisfying $\mathcal{D}_{\text{f}}$, there is a strict reduction relation of $\Delta^{\text{wheels}}_{\text{fluent}}$. Consider the following binary relation $\prec_1$:

$$\left\{ \begin{array}{l} (I_{T_i}, T_i(S_0^I)) \\ (T_i(s), T_i(do^I(a,s)) \\ (C_{T_i}(a,s), T_i(do^I(a,s)) \\ (C_{\neg T_i}(a,s), T_i(do^I(a,s)) \end{array} \middle| \quad i = 1,2 \wedge a \in Act^I \wedge s \in Sit^I \right\}$$

It expresses that $T_i(S_0)$ depends on its initial predicate, and $T_i(do^I(a,s))$ depends on $T_i(s), C_{T_i}(a,s)$ and $C_{\neg T_i}(a,s)$. It is easy to verify that $\prec_1$ is a reduction relation in I and that its transitive closure is a strict reduction relation of $\Delta^{\text{wheels}}_{\text{fluent}}$. Then Theorem 4 guarantees that Δ_{fluent} is total in I and that Δ_{fluent} and $comp(\Delta_{\text{fluent}})$ are equivalent in I.

Next, consider that $\Delta^{\text{wheels}}_{\text{effect}}$ is a positive inductive definition, so by Theorem 3, it is total in each structure and it is equivalent to $PID(\Delta^{\text{wheels}}_{\text{effect}})$.

We have shown now that given $\mathcal{D}_{\text{f}}$ the component definitions $\Delta^{\text{wheels}}_{\text{fluent}}$ and $\Delta^{\text{wheels}}_{\text{effect}}$ can be translated to the appropriate classical logic axioms. It remains to be shown that $\Delta^{\text{wheels}}_{\text{sc}}$ can be split in the conjunction $\Delta^{\text{wheels}}_{\text{fluent}} \wedge \Delta^{\text{wheels}}_{\text{effect}}$. Consider the following binary relation:

$$\prec_2 := \prec_1 \cup \left\{ \begin{array}{l} (C_{T_1}(a,s), C_{T_2}(a,s)) \\ (C_{T_2}(a,s), C_{T_1}(a,s)) \\ (C_{\neg T_1}(a,s), C_{\neg T_2}(a,s)) \\ (C_{\neg T_2}(a,s), C_{\neg T_1}(a,s)) \end{array} \middle| \quad a \in Act^I \wedge s \in Sit^I \right\}$$

It is easy to verify that $\prec_2$ is a reduction relation of $\Delta^{\text{wheels}}_{\text{sc}}$ and, since each superset of a reduction relation is obviously also a reduction relation, its transitive reflexive closure $\prec_2^*$ is also a reduction relation. Also, one can verify that $\prec_2^*$ is a pre-well-order on At^τ_A. Moreover, no causality atom depends on a fluent atom according to $\prec_2^*$. Consequently, if $P[\bar{a}]$ is a fluent atom and $Q[\bar{b}]$ a causation atom, then $P[\bar{a}] \not\prec_2^* Q[\bar{b}]$ and $Q[\bar{b}] \prec_2^* P[\bar{a}]$ implies $Q[\bar{b}] \prec\!\!\!\prec_2^* P[\bar{a}]$. We already showed that $\Delta^{\text{wheels}}_{\text{fluent}}$ and $\Delta^{\text{wheels}}_{\text{effect}}$ are total in each structure satisfying $\mathcal{D}_{\text{f}}$. It follows that $\{\Delta^{\text{wheels}}_{\text{fluent}}, \Delta^{\text{wheels}}_{\text{effects}}\}$ is a total reduction partition of $\Delta^{\text{wheels}}_{\text{sc}}$ in each structure satisfying $\mathcal{D}_{\text{f}}$. We can apply Theorem 2 and the proposition follows.

5 Conclusions

Recently, we argued [Den00,DBM01] that non-monotone forms of inductive definitions such as iterated inductive definitions and definitions over well-orders, can play a unifying role in logic, AI and knowledge representation, connecting remote areas such as non-monotonic reasoning, logic programming, description logics, deductive databases and fixpoint logics. In this paper, we further substantiated this claim by defining a more general logic integrating classical logic and monotone and non-monotone inductive definitions and applying it to what has always been the most important test domain of knowledge representation — temporal reasoning.

The main technical theorems here are the Modularity theorem and the theorems translating certain classes of NMID-definitions into classical logic formalisations. Problem-free composition is crucial while axiomatizing a complex system. Because definitions in our logic are non-monotone, composing or decomposing definitions is in general not equivalence preserving. However, the conditions we have presented allow one to separate problem-free (de)compositions from those causing change in meaning. We have shown that the Modularity theorem is useful also for analyzing complex definitions — some properties of large definitions are implied by properties of sub-definition. The Modularity theorem is also an important tool for simplifying logical formulas with definitions by translating them into formulas of classical logic.

As an application of our logic, we presented the inductive situtation calculus. In this formalisation, the situation calculus is modeled as an inductive definition, defining fluents and causality predicates by simultaneous induction in the

well-ordered set of situations. An important aspect of our formalisation is that causation rules can be represented in a modular way by rules in an inductive definition. We used the Modularity theorem to demonstrate its equivalence with a situation calculus axiomatization based on completion and circumscription. In a forthcoming paper, we present a general solution to the ramification problem for the situation calculus within the NMID-logic.

Future Work Given the connection between the μ-calculus and inductive definitions, it would be interesting to study NMID-logic on Kripke structures, and to understand what form the splitting conditions take on these structures. This may lead to new results on compositional verification. Imagine a complex system consisting of numerous inter-related components (e.g. gear wheels). A formal description of the behavior of such a system over time requires mutual recursion between its modules. If the specification can be decomposed into specifications of individual parts, then the complexity of verification of the entire system will be reduced since one has to deal with a reduced state space each time. Other directions include the study of the connections of our logic with the work on iterated induction by Feferman and others, and also applications of our modularity techniques for simplifying formulas with definitions through their translation to first- and second-order logic.

References

[Acz77] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. Elsevier, 1977.

[DBM01] M. Denecker, M. Bruynooghe, and V. Marek. Logic programming revisited: Logic programs as inductive definitions. *ASM Transactions on Computational Logic (TOCL)*, 4(2), 2001.

[Den98] M. Denecker. The well-founded semantics is the principle of inductive definitions. In J. Dix, L. Farinas del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence*, volume 1489 of *Lecture Notes in Artificial Intelligence*, pages 1–16. Springer-Verlag, 1998.

[Den00] M. Denecker. Extending classical logic with inductive definitions. In *Proc. CL'2000*, 2000.

[DG02] A. Dawar and Y. Gurevich. Fixed point logics. *The Bulletin of Symbolic Logic*, 8(1):65–88, 2002.

[DMT00] M. Denecker, V. Marek, and M. Truszczynski. Approximating operators, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In J. Minker, editor, *Logic-Based AI*, pages 127–144. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[DTV98] M. Denecker, D. Theseider Duprè, and K. Van Belleghem. An inductive definition approach to ramifications. *Linköping Electronic Articles in Computer and Information Science*, 3(1998): nr 7, 1998. URL: http://www.ep.liu.se/ea/cis/1998/007/.

[Fit03] M. Fitting. Fixpoint semantics for logic programming - a survey. *Theoretical Computer Science*, 2003. To appear.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Mos74] Y.N. Moschovakis. *Elementary Induction on Abstract Structures*. Amsterdam, North Holland, 1974.

[Rei01] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems.* MIT Press, 2001.

[Sch95] J. Schlipf. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences*, 51:64–86, 1995.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[Van93] A. Van Gelder. An alternating fixpoint of logic programs with negation. *Journal of computer and system sciences*, 47:185–221, 1993.

[VDD00] S. Verbaeten, M. Denecker, and D. De Scheye. Compositionality of normal open logic programs. *journal of Logic Programming*, 1(3):151–183, 2000.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of Assoc. Comput. Mach.*, 38(3):620–650, 1991.

Reasoning About Actions and Change in Answer Set Programming

Yannis Dimopoulos[1], Antonis C. Kakas[1], and Loizos Michael[2]

[1] Department of Computer Science, University of Cyprus
P.O. Box 20537, CY1678, Nicosia, Cyprus
yannis,antonis@cs.ucy.ac.cy
[2] Division of Engineering and Applied Sciences, Harvard University
33 Oxford Str., Cambridge, MA 02138, USA
loizos@eecs.harvard.edu

Abstract. This paper studies computational issues related to the problem of reasoning about actions and change (RAC) by exploiting its link with the Answer Set Programming paradigm. It investigates how increasing the expressiveness of a RAC formalism so that it can capture the three major problems of frame, ramification and qualification, affects its computational complexity, and how a solution to these problems can be implemented within Answer Set Programming. Our study is carried out within the particular language $\mathcal{E}$. It establishes a link between language $\mathcal{E}$ and Answer Set Programming by presenting encodings of different versions of this language into logic programs under the answer set semantics. This provides a computational realization of solutions to problems related to reasoning about actions and change, that can make use of the recent development of effective systems for Answer Set Programming.

1 Introduction

Reasoning about actions and change (RAC) is an important topic in Artificial Intelligence with several proposals of how to address the problem. In particular, the authors of [5] have proposed the use of high-level action description languages intended as specification languages that are to be computed via a translation to some general-purpose formalism such as logic programming. A recent such language is Language $\mathcal{E}$ [12] based on an Event Calculus ontology. Language $\mathcal{E}$ extends the Event Calculus in various ways and provides solutions to the ramification and qualification problems. In particular, for the qualification problem it uses the notion of *default actions* and associated *default effect laws* [10].

This paper studies, in the above spirit of introducing action description languages, a link between language $\mathcal{E}$, and more generally RAC, and the new declarative logic programming approach of *Answer Set Programming* (ASP) [15]. It presents various ASP encodings of reasoning problems in language $\mathcal{E}$ by exploiting the close relation between the underlying model theoretic semantics of $\mathcal{E}$ and the answer set (stable model) semantics of logic programming. This translation to ASP provides a principled method for computing in the language $\mathcal{E}$ that

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 61–73, 2004.

can make use of the development of effective systems for ASP. Indeed, in recent years there has been remarkable progress in the development of effective systems that compute answer sets, eg. `DLV` [3] and `smodels` [19], and their application to various problems such as planning, diagnosis, and model checking. Our work therefore continues a long history of strong links between RAC and logic programming, starting in the mid 80s with the proposal of the Event Calculus. In this sense it has close links with other more recent works, such as [1,2,4,5,6,7,17, 18], which use logic programming (many of which within ASP) in their study of RAC problems.

In our work we have, in addition, tried to address various computational issues of the frame, ramification and qualification problems in a systematic way where we incrementally consider the synthesis of these problems. These three major problems of RAC are associated to three different levels of expressivity in the language $\mathcal{E}$: the basic language $\mathcal{E}$ (with strict action effect laws) covering the frame problem, inclusion of ramification statements for the ramification problem and finally allowing the action effect laws to be default for the qualification problem. Our study shows that normal logic programs under the stable model semantics can encode theories in $\mathcal{E}$ with strict actions and ramifications. Moreover, if no ramifications are present, the logic programming encoding is stratified (for a complete initial state) and therefore the corresponding RAC semantics is polynomial.

Furthermore, when we extend the language $\mathcal{E}$ with default actions, it is shown that reasoning becomes Σ_2^P-hard. In this case, finding a model for a theory, requires the use of two normal programs, a generator and a tester, as it has been proposed for problems of similar complexity in [8].

2 Preliminaries for Language $\mathcal{E}$

In this section, we review briefly the language $\mathcal{E}$ [12] and its extensions to address the ramification [11] and qualification [10] problems. We will consider only a simplified form of the language, but sufficient to encode these problems.

A domain description or theory in $\mathcal{E}$ is a finite collection of the following types of statements (where A is an action constant, T is a time point – where for simplicity we assume that it belongs to the natural numbers, F is a fluent constant or atom, L is a fluent literal and C is a set of fluent literals):

- *t-propositions* of the form: L `holds-at`T
- *h-propositions* of the form: A `happens-at`T
- *c-propositions* of the form: A `initiates` F `when`C or, A `terminates` F `when`C
- *r-propositions* of the form: L`whenever`C.

R-propositions, L `whenever` C, are also called **ramifications** and serve a dual role in that they describe both static constraints (as classical implications) between fluents and ways in which fluents may be indirectly affected by action occurrences. They encode a notion of *causality* in the theory from the conditions C to L in the sense that "every action occurrence that brings about C also brings about L". The following example illustrates the syntax of $\mathcal{E}$.

Example 1. (Car Domain: D_c)
$TurnOnKey$ `initiates` $Running$ `when` $\{Battery\}$
$TurnOffKey$ `terminates` $Running$
$\neg Running$ `whenever` $\{Broken\}$
$\neg Running$ `whenever` $\{\neg Petrol\}$
$Battery$ `holds-at` 0
$TurnOnKey$ `happens-at` 2

The semantics of $\mathcal{E}$ assumes that no events occur other than those explicitly given in D and is based on a notion of a *model* of a domain D. Consider first the case where D contains no r-propositions.

Definition 1. *Let D be a theory in the language $\mathcal{E}$. A map, $H : \Phi \times \Pi \mapsto \{true, false\}$, where Φ is the set of fluent constants and Π the set of time constants, is an interpretation of D. Given a time point T and a fluent constant F, we say that T is an* **initiation-point** *(resp.,* **termination-point***)* **for** *F* **in** *H* **relative to** *D iff there is an action constant A such that (i) D contains both an h-proposition "A* `happens-at` *T" and a c-proposition "A* `initiates` *(resp.,* `terminates`*) F* `when` *C", and (ii) H satisfies C at T (i.e. for each $F \in C$, $H(F,T) = true$, and for each F such that $\neg F \in C$, $H(F,T) = false$).*

In the example above, 2 is an initiation-point for $Running$ in any interpretation that has $Battery$ true at 2. When a domain D contains r-propositions this definition has to be extended to allow for initiation or termination points that are generated recursively through these r-propositions with the above definition as the base case. The ramification statements thus give indirect initiation/termination points and hence generate indirect effect laws. The formal definition is given below.

Definition 2. *Let D be a theory in the language $\mathcal{E}$, and H an interpretation of D. Let $\mathcal{W}$ be the set $2^{\Phi\times\Pi} \times 2^{\Phi\times\Pi}$ and let the operator $\mathcal{F} : \mathcal{W} \mapsto \mathcal{W}$ be defined as follows. For each, $\langle \mathcal{I}n, \mathcal{T}e\rangle \in \mathcal{W}$ denote $\mathcal{F}(\langle \mathcal{I}n, \mathcal{T}e\rangle)$ by $\langle \mathcal{I}n', \mathcal{T}e'\rangle$. Then for any $F \in \Phi$ and $T \in \Pi$, (F,T) is in $\mathcal{I}n'$ (resp., in $\mathcal{T}e'$) iff one of the following two conditions holds.*

1. *There is an action constant A such that (i) there is both an h-proposition in D of the form "A* `happens-at` *T" and a c-proposition in D of the form "A* `initiates` *F* `when` *C" (resp., "A* `terminates` *F* `when` *C") and (ii) H satisfies C at T.*
2. *There is an r-proposition in D of the form "F* `whenever` *C" (resp., "$\neg F$* `whenever` *C") and a partition $\{C_1, C_2\}$ of C such that (i) C_1 is non-empty, for each fluent constant $G \in C_1$, $(G,T) \in \mathcal{I}n$, and for each fluent literal $\neg G \in C_1$, $(G,T) \in \mathcal{T}e$, and (ii) there is some $T_2 \in \Pi$, $T \prec T_2$, such that for all T_1, $T \prec T_1 \preceq T_2$, H satisfies C at T_1.*

Let $\langle \mathcal{I}n^f, \mathcal{T}e^f\rangle$ be the least fixed point of the (monotonic) operator $\mathcal{F}$ starting from the empty tuple $\langle \emptyset, \emptyset\rangle$. T is an initiation-point (resp., termination-point) for F in H relative to D iff $(F,T) \in \mathcal{I}n^f$ (resp., $(F,T) \in \mathcal{T}e^f$).

A model is then defined as follows.

Definition 3. *Let D be a theory in the language $\mathcal{E}$. Then an interpretation H is a* **(strict) model** *of D iff, for every fluent constant F and time-points $T_1 \prec T_3$:*

1. *If there is no initiation (resp., termination) point T_2 for F in H such that $T_1 \preceq T_2 \prec T_3$, then $H(F,T_1) = false \Rightarrow H(F,T_3) = false$ (resp., $H(F,T_1) = true \Rightarrow H(F,T_3) = true$) i.e. fluents change their truth values only via occurrences of initiating or terminating actions.*
2. *If T_1 is an initiation-point for F in H, and there is no termination-point T_2 for F in H such that $T_1 \prec T_2 \prec T_3$, then $H(F,T_3) = true$, i.e. initiating a fluent establishes its truth value as true.*
3. *If T_1 is a termination-point for F in H, and there is no initiation-point T_2 for F in H such that $T_1 \prec T_2 \prec T_3$, then $H(F,T_3) = false$, i.e. terminating a fluent establishes its truth value as false.*
4. *H satisfies the following constraints:*
   - *For all "F `holds-at` T" in D, $H(F,T) = true$, and for all "$\neg F$ `holds-at` T'" in D, $H(F,T') = false$.*
   - *For all "L `whenever` C" in D, and time-points T, if H satisfies C at T then H satisfies L at T.*

A theory D **entails credulously** the t-proposition F `holds-at` T (resp., $\neg F$ `holds-at` T), iff there exists a model H of D such that $H(F,T) = true$ (resp., $H(F,T) = false$). If the t-proposition is satisfied in every model of D we say that it is **entailed skeptically**.

All models of the previous example domain make *Battery* true at all time points due to its persistence (first condition) from time 0 (fourth condition). From time 3 onwards *Running* necessarily holds in all models (second condition) as this is initiated by the $TurnOnKey$ action at time 2 and it is not terminated at any later time point. Note that then the ramification constraints impose (fourth condition) that *Broken* is false from 3 onwards and since 2 is not a termination point for *Broken*, it is false at all previous time points as well (first condition).

The above definition of a model handles the *frame problem* through the default persistence that its first condition imposes. It also handles the *ramification problem* through the production of additional initiation and termination points generated by r-propositions as described in definition 2 above. To handle also the *qualification problem*, we consider the action effect laws as default so that although an action is executed this may not produce all of its effects. For this we relax the second and third conditions of a model from absolute requirements to hold only if the other requirements of the first and fourth conditions are not violated. This default semantics of the language is denoted by $\mathcal{E}_{\mathcal{D}}$.

Definition 4. *Let D be a theory in the common syntax of languages $\mathcal{E}$ and $\mathcal{E}_{\mathcal{D}}$. Then an interpretation H is a* **pre-model** *of D in $\mathcal{E}_{\mathcal{D}}$ iff this satisfies the first and fourth conditions of the definition 3 for a model in the (strict) language $\mathcal{E}$.*

For theories that contain *complete initial states*, the simple preference relation of minimizing *failed actions* in their pre-models captures their intended meaning.

Definition 5. *Given a theory D in the language $\mathcal{E}_\mathcal{D}$ with a complete initial state, a pre-model M of D, and a default action A such that D contains the propositions "A* `initiates` *(resp.,* `terminates`*) F* `when` *C" and "A* `happens-at` *T", we say that A* **fails** *in M at time T wrt F (resp., $\neg F$), if C is true in M at T and F is false (resp. true) in M at $T+1$. We then say that the triple (A, F, T) (resp., $(A, \neg F, T)$) is* **failed** *in M, and denote by $F(M)$ the set of failed triples of M.*

We then define the set of (default) models of such a domain, and hence the semantics of $\mathcal{E}_\mathcal{D}$, as follows.

Definition 6. *Given a theory D in the language $\mathcal{E}_\mathcal{D}$ with a complete initial state, and a pre-model M of D, we say that M is a* **(default) model** *of D, iff there is no other pre-model M' of D such that $F(M') \subset F(M)$.*

We note that the above semantics can not be extended to $\mathcal{E}_\mathcal{D}$ theories with incomplete initial states by simply extending the preference relation on pre-models over all possible completions of these theories. Consider for instance the following simple incomplete theory

Break `initiates` *Broken*
TurnOnKey `initiates` *Running* `when` {*Battery*}
¬*Running* `whenever` {*Broken*}
Break `happens-at` 1
TurnOnKey `happens-at` 2

Consider all possible value assignments to atoms *Broken*, *Battery*, *Running* and their associated pre-models. The definition 6 above, yields a unique model that assigns *Battery* the value false at every time point, which is a rather counterintuitive result. Such problems do not arise if the initial state is complete. Correct treatment of theories with incomplete initial states necessitates a more involved semantics that will be discussed in a longer version of this paper.

We also note that the default models introduced above are called **non-chronological** models. In some cases, we may want to consider only their subset of **chronological** models where the earlier action occurrences succeed to produce their effects over strictly later occurring actions. The details of these chronological models are not important for this paper.

Let us consider again our running example and now interpret this in $\mathcal{E}_\mathcal{D}$ instead of $\mathcal{E}$, i.e. consider its c-propositions as defeasible default effect laws rather than strict laws. Suppose that we complete the initial state by adding the t-propositions ¬*Running* `holds-at` 0, *Petrol* `holds-at` 0 and ¬*Broken* `holds-at` 0. This theory has the same meaning as under the strict action semantics, where *Running* holds skeptically from time 3 onwards etc. Suppose now that we replace ¬*Broken* `holds-at` 0 by *Broken* `holds-at` 0. Then under $\mathcal{E}_\mathcal{D}$ we have that ¬*Running* holds everywhere, with the occurrence of *TurnOnKey* at 2 failing to produce its effect, whereas under $\mathcal{E}$ the theory becomes inconsistent. Similarly, if we add instead the two sentences *Break* `initiates` *Broken* and *Break* `happens-at` 1, under $\mathcal{E}$ the theory will become inconsistent whereas under $\mathcal{E}_\mathcal{D}$ we will have two (non-chronological) default models. In the first model *Break* succeeds at 1 and so *Broken* is true from time 2 onwards and hence the

second action $TurnOnKey$ is forced to fail to produce its effect, thus $Running$ is false. In the second model the occurrence of $Break$ fails to produce its effect whereas $TurnOnKey$ succeeds and so $Running$ holds from 3 onwards. Note that we cannot have a model where both actions produce their effects. If we are interested only in chronological models then we have only the first model.

3 Translating $\mathcal{E}$ into Answer Sets

In this section we present encodings of various versions of language $\mathcal{E}$ into stable models. Throughout the paper we assume that *strict* actions that are executed concurrently do not have contradictory effects, essentially excluding the possibility of trivially inconsistent theories. In the rest of this paper, for each fluent F, F' is used to denote the negation of F. The relation between the two symbols is captured through the constraint $\leftarrow F(T), F'(T)$, which is included in all encodings that are described in the paper.

3.1 Strict Actions

We start our study by examining theories in language $\mathcal{E}$ that have a complete initial state and do not contain any r-propositions. Such theories can be translated in normal logic programs as follows.

- every t-proposition of the form F `holds-at` T, translates into $F(T)$.
- every t-proposition of the form $\neg F$ `holds-at` T, translates into $F'(T)$.
- every h-proposition of the form A `happens-at` T translates into $A(T)$.
- every c-proposition of the form A `initiates` F `when` C translates into $generated_F(T+1) \leftarrow A(T), C(T)$.
- every c-proposition of the form A `terminates` F `when` C translates into $generated_{F'}(T+1) \leftarrow A(T), C(T)$.

In addition to the above, for each atom F in the theory the logic program contains the following rules.

- the pair of rules $F(T) \leftarrow generated_F(T)$ and $F'(T) \leftarrow generated_{F'}(T)$.
- the frame axioms

$$F(T+1) \leftarrow F(T), not\ generated_{F'}(T+1)$$
$$F'(T+1) \leftarrow F'(T), not\ generated_F(T+1).$$

We denote by $G_E^S(H)$ the logic program that results from the above translation of an $\mathcal{E}$ theory H. Now assume that an $\mathcal{E}$ theory contains an initial state that is not complete, in the sense that some fluents are not assigned a truth value at that state. For each incomplete fluent F, the logic programming encoding is extended with the rules $F(0) \leftarrow not\ F'(0)$ and $F'(0) \leftarrow not\ F(0)$.

Although the semantics of the language $\mathcal{E}$ has been developed outside answer set programming, it has similar model theoretic features as that of the answer set semantics and hence it is relatively easy to show a direct correspondence between an $\mathcal{E}$ theory and its translated program, as stated in the following theorem.

Theorem 1. *Let H be an $\mathcal{E}$ theory over the linear time structure of the natural numbers without any ramification statements and $G_E^S(H)$ its translated logic program. Then M is a model of H iff S_M is an answer set of $G_E^S(H)$, where for any fluent F, action constant A and time point T,*

- $A(T) \in S_M$ *iff* $A(T) \in G_E^S(H)$.
- $F(T) \in S_M$ *(resp.,* $F'(T) \in S_M$*) iff* $M(F,T) = true$ *(resp.,* $false$*).*
- $generated_K(T+1) \in S_M$*, where* $K = F$ *or* $K = F'$*, iff* $G_E^S(H)$ *contains a rule* $generated_K(T+1) \leftarrow A_i(T), C_i(T)$ *such that* $A_i(T) \in G_E^S(H)$*, and* $C_i(T)$ *is satisfied by* M*.*

In the special case where the $\mathcal{E}$ theory H has a complete initial state we can use the previous theorem to obtain the following result.

Corollary 1. *Let H be an $\mathcal{E}$ theory with a complete initial state and without any ramification statements. Then H has at most one $\mathcal{E}$ model and finding this has polynomial complexity.*

The result holds because the logic program $G_E^S(H)$ is stratified. A simple method for computing its stable model is to find first its well-founded model, and then check whether this satisfies the constraints $\leftarrow F(T), F'(T)$. A similar result has been obtained in [14] for the language $\mathcal{A}$. Our translation extends this result by showing that any ASP solver that employs well-founded computation in its constraint propagation algorithm, is able to solve the problem in polynomial time.

It turns out however that, if the initial state is incomplete, finding a model of an $\mathcal{E}$ theory is an NP-hard problem. Again, a similar result has been proven for language $\mathcal{A}$ in [14].

Theorem 2. *Deciding whether an $\mathcal{E}$ theory with an incomplete initial state and without any ramification statements has a model is NP-hard.*

3.2 Strict Actions and Ramifications

Suppose now that the theory contains some r-propositions. Propositions of the form $false$ `whenever` C translate into the constraint $\leftarrow C(T)$. A proposition of the form F `whenever` C translates into $F(T) \leftarrow C(T)$, while $\neg F$ `whenever` C into $F'(T) \leftarrow C(T)$. Furthermore, the presence of r-propositions requires a modification in the definition of $generated_K(T)$, to account for the indirect generation of effects through the ramifications. For every r-proposition L `whenever` $\{C_1, ..., C_n\}$, where L is a literal, and for every $i : 1 \leq i \leq n$ we add the rule

$$generated_L(T) \leftarrow C_1(T), ..., C_n(T), generated_{C_i}(T),$$

where $generated_K(T)$ denotes $generated_F(T)$ when $K = F$, and $generated_{F'}(T)$ when $K = \neg F$, for some atom F. We denote by $G_E(H)$ the logic program that corresponds to an $\mathcal{E}$ theory H that contains ramifications.

Example 2. Consider the theory H_1
A_1 `initiates` F_2 A_2 `terminates` F_2 F_1 `whenever` $\{F_2\}$
A_1 `happens-at` 2 A_2 `happens-at` 4 $\neg F_1$ `holds-at` 5
The program $G_E(H_1)$ of this theory contains the rules
$A_1(2)$ $A_2(4)$ $F_1'(5)$
$generated_{F_2}(T+1) \leftarrow A_1(T)$ $generated_{F_2'}(T+1) \leftarrow A_2(T)$
$F_1(T) \leftarrow F_2(T)$ $generated_{F_1}(T) \leftarrow F_2(T), generated_{F_2}(T)$
$F_1(T) \leftarrow generated_{F_1}(T)$ $F_1'(T) \leftarrow generated_{F_1'}(T)$
$F_2(T) \leftarrow generated_{F_2}(T)$ $F_2'(T) \leftarrow generated_{F_2'}(T)$
$F_1(T+1) \leftarrow F_1(T), not\ generated_{F_1'}(T+1)$
$F_1'(T+1) \leftarrow F_1'(T), not\ generated_{F_1}(T+1)$
$F_2(T+1) \leftarrow F_2(T), not\ generated_{F_2'}(T+1)$
$F_2'(T+1) \leftarrow F_2'(T), not\ generated_{F_2}(T+1)$
The above program derives both $F_1(5)$ and $F_1'(5)$ and therefore is inconsistent.

The corollary below extends theorem 1 to the case where the $\mathcal{E}$ theory is allowed to contain ramification statements.

Corollary 2. *Let H be an $\mathcal{E}$ theory over the linear time structure of the natural numbers possibly containing ramification statements and $G_E(H)$ the extended translated logic program of H. Then the same correspondence between the $\mathcal{E}$ models of H and the consistent answer sets of $G_E(H)$ holds as in theorem 1.*

The next result states that ramifications increase the complexity of reasoning when the initial state is complete. Note that, under a complete initial state, an $\mathcal{E}$ theory without ramifications has at most one model, whereas theories with ramifications may have more than one model.

Theorem 3. *Deciding whether an $\mathcal{E}$ theory with a complete initial state and ramifications has a model is NP-hard.*

3.3 Default Actions

In this subsection we describe how an $\mathcal{E}_\mathcal{D}$ theory can be translated into answer set programming. Before we proceed with the translation we present two complexity results indicating that reasoning with an $\mathcal{E}_\mathcal{D}$ theory is, in the general case, computationally harder than with an $\mathcal{E}$ theory. All the following results that refer to default theories, concern theories with a complete initial state.

Theorem 4. *Deciding whether a pre-model of an $\mathcal{E}_\mathcal{D}$ theory H with a complete initial state is a model of H is coNP-hard.*

Proof (sketch): We use a reduction from the problem of deciding whether a 3-CNF formula is unsatisfiable. Given a 3-CNF formula C we construct an $\mathcal{E}_\mathcal{D}$ theory H_C such that the value assignment M (defined below) is a model of H_C iff C is unsatisfiable.
For each atom a_i of C, H_C contains atoms a_i^+ and a_i^-, representing the literals a_i and $\neg a_i$ respectively. The theory H_C also contains the atom m and an atom C_i for each clause of C. The propositions that are included in H_C are the following

- the r-propositions C_i `whenever` $\{h_{l_{ij}}, m\}$, $1 \leq j \leq 3$ for each clause C_i of C of the form $C_i = l_{i1} \vee l_{i2} \vee l_{i3}$, where atom $h_{l_{ij}}$ represents the literal l_{ij}.
- the r-propositions $false$ `whenever` $\{\neg C_i, m\}$, for each clause C_i of C.
- the r-propositions $false$ `whenever` $\{a_i^+, a_i^-, m\}$ and the pair of propositions a_i^+ `whenever` $\{\neg a_i^-, m\}$ and a_i^- `whenever` $\{\neg a_i^+, m\}$, for each atom a_i.
- the default effect rule A `initiates` m and the proposition A `happens-at` 1.
- the initial state is completely determined by the propositions $\neg m$ `holds-at` 0, $\neg C_i$ `holds-at` 0, $\neg a_i^+$ `holds-at` 0, and $\neg a_i^-$ `holds-at` 0.

Define M to be an interpretation that assigns *false* to all fluents at all time-points. It is not hard to verify that M is a pre-model of H_C and has one failed action. We will show that M is a model of H_C iff C is unsatisfiable.
Assume that C is satisfiable. Then, there exists a truth assignment to the atoms of C that satisfies every clause of C. Consider the interpretation M' of H_C that assigns: (a) *false* to all fluents, at time-points 0 and 1; (b) *true* to fluents m and C_i, at all time-points from 2 onwards; (c) *true* to one and *false* to the other atom in each pair a_i^+ and a_i^-, according to the satisfying assignment of C, at all time-points from 2 onwards. It can be proven that M' is a pre-model of H_C and since action A does not fail in M', then M cannot be a model of H_C.
Assume that M is not a model of H_C. Then, there exists a pre-model M' of H_C in which action A does not fail and gives m. Consequently, every C_i is true at time 2 and false at time 0. In order for M' to explain the change, every C_i must be an indirect effect of action A. For this to happen, by construction of H_C the effects a_i^+ or a_i^- of action A must necessarily induce a satisfying truth assignment for C, and therefore C is satisfiable. □

Theorem 5. *Let H be an $\mathcal{E}_\mathcal{D}$ theory with a complete initial state and Q a query of the form q* `holds-at` *T. The problem of deciding whether Q is a credulous conclusion of H is Σ_2^P-hard.*

Proof (sketch): We use a reduction from the problem of deciding whether, a literal l is true in some minimal model of a given 3-CNF formula C. Assume that C has n atoms and k clauses. We construct an $\mathcal{E}_\mathcal{D}$ theory H_C such that l is a credulous conclusion of H_C iff l is true in a minimal model of C.
Each clause $C_i = l_{i1} \vee l_{i2} \vee l_{i3}$ of C, translates into three r-propositions C_i `whenever` $\{l_{ij}, m\}$, $1 \leq j \leq 3$, where m and C_i are atoms that do not occur in C. For every atom a_i of C, H_C contains a default action A_i, and the c-proposition A_i `terminates` a_i. Let $A_{i1}, A_{i2}, ..., A_{in}$ be any permutation of the set of actions $A_1, ..., A_n$. We add to H_C the set of h-propositions of the form, A_{ij} `happens-at` j. Moreover, we add the r-proposition F `whenever` $\{C_1, ..., C_k\}$ and the observation F `holds-at` $n+2$. Finally, H_C contains the propositions A^m `initiates` m and A^m `happens-at` $n+1$, for some distinct action A^m. The initial state of H_C is completely determined by the observations a_i `holds-at` 0, $\neg m$ `holds-at` 0, $\neg F$ `holds-at` 0 and $\neg C_i$ `holds-at` 0. It can be shown that l is true in some minimal model of C iff the proposition l `holds-at` $n+2$ is a credulous $\mathcal{E}_\mathcal{D}$ conclusion of the theory H_C. □

3.4 Computing $\mathcal{E}_\mathcal{D}$

Our approach to finding a model of an $\mathcal{E}_\mathcal{D}$ theory is to first compute a pre-model of the theory, and then repeatedly try to improve it until it becomes a model. To implement this, we use a combination of a generator and a tester program in the same vein of [8]. The generator computes the stable models of a program that correspond to the pre-models of the theory. The tester checks whether these are minimal wrt the failed triples. If the stable model is not minimal, the tester produces a better candidate.

We first describe how, given an $\mathcal{E}_\mathcal{D}$ theory H we obtain a logic program $G_{E_D}(H)$ such that the answer sets of $G_{E_D}(H)$ are in direct correspondence with the pre-models of H. The translation is identical to $G_E(H)$ for strict actions and ramifications, described in subsection 3.2, with the difference that for each rule $generated_K(T+1) \leftarrow A(T), C(T)$, where $K = F$ or $K = F'$ for some atom F,

- we introduce two new predicates $OnA(K,T)$ and $OffA(K,T)$.
- we replace the rule $generated_K(T+1) \leftarrow A(T), C(T)$ in $G_E(H)$ with the rule $generated_K(T+1) \leftarrow A(T), C(T), OnA(K,T)$.
- we add a set of choice rules of the form

$$OnA(K,T) \leftarrow not\ OffA(K,T) \qquad OffA(K,T) \leftarrow not\ OnA(K,T).$$

The above changes account for the fact that actions can arbitrarily produce or not their effects in a pre-model even in the case when their pre-conditions, C, are satisfied. Therefore, when a triple (A,F,T) (resp. $(A,\neg F,T)$) is failed in a pre-model then the predicate $OffA(F,T)$ (resp., $OffA(F',T)$) is assigned true in the corresponding answer set. Given the above, it is easy to show, in a way analogous to the proof of theorem 1, that the pre-models of an $\mathcal{E}_\mathcal{D}$ theory H are in one-to-one correspondence with the stable models of the translated program $G_{E_D}(H)$. Hence $G_{E_D}(H)$ can be used as the generator for a given $\mathcal{E}_\mathcal{D}$ theory H. The tester then needs to produce a stable model whose corresponding pre-model is preferred over the pre-model corresponding to the stable model given by the generator. One way to implement such a tester, following directly from the preference relation implied by definition 6, is to use the logic program $T_E(H,M) = G_{E_D}(H) \cup N(H,M)$, where given a generated model M to be tested, $N(H,M)$ consists of the rules:

- $failed(A_i,F,T) \leftarrow$ for every failed triple (A_i,F,T) of M.
- $failed(A_i,F',T) \leftarrow$ for every failed triple $(A_i,\neg F,T)$ of M.
- $better \leftarrow A_i(T), C_i(T), F(T+1), failed(A_i,F,T)$, for every c-proposition of the form A_i `initiates` F `when` C_i and h-proposition A_i `happens-at` T, and $better \leftarrow A_i(T), K_{ij}(T), failed(A_i,F,T)$, for each $L_{ij} \in C_i$, where if $L_{ij} = F_{ij}$ then $K_{ij} = F'_{ij}$ or if $L_{ij} = \neg F_{ij}$ then $K_{ij} = F_{ij}$.
  Similarly, for each c-proposition A_i `terminates` F `when` C_i.
- $worse \leftarrow A_i(T), C_i(T), F'(T+1), not\ failed(A_i,F,T)$ for every c-proposition of the form A_i `initiates` F `when` C_i and h-proposition A_i `happens-at` T.
  Similarly, for each c-proposition A_i `terminates` F `when` C_i.
- the constraints $\leftarrow not\ better$ and $\leftarrow worse$.

A stable model of the tester $T_E(H, M)$ improves on M with fewer (i.e. a subset) triples that fail. This is ensured by the constraints $\leftarrow$ *not better* and $\leftarrow$ *worse* (the latter ensures that all triples that have not failed in M also do not fail in the new stable model, whereas the former constraint ensures that at least one new triple that failed in the given stable model does not fail in the new stable model). This improved stable model is then fed back to the tester until this fails to have a stable model.

In this paper our analysis is restricted to $\mathcal{E}_\mathcal{D}$ theories with a complete initial state. When the initial state is not complete, the definition of the models of $\mathcal{E}_\mathcal{D}$ can no longer rely on the simple minimization of failures used in the complete initial state case. One way to address this extension would be to first define what constitutes a sufficient reason for an action to fail, given the uncertainty of the initial state, and then apply a preference amongst all the pre-models that minimizes "unexplained" failures. Hence, although at the top level we would still have a simple minimization, and we could thus use ASP in a similar way, what is being actually minimized is not easily defined, and in fact might require an additional minimization to be computed. This not only makes the semantics of $\mathcal{E}_\mathcal{D}$ more involved, and hence the translation to ASP more complex, but the resulting translation might also have a different computational complexity than the one for the complete initial state case.

4 Reasoning in Language $\mathcal{E}$

The study so far has been restricted to the problem of finding the models of some $\mathcal{E}$ theory. This section discusses how the encodings described in the previous section can be used to implement various reasoning tasks in language $\mathcal{E}$.

The treatment of strict $\mathcal{E}$ theories, where their models are captured by a single logic program, is straightforward. To decide whether a query $Q = p$ `holds-at` T is a credulous conclusion of a theory H we simply search for stable models that contain Q. Checking whether Q is a skeptical conclusion requires deciding whether Q is a credulous conclusion and $Q' = \neg p$ `holds-at` T is not a credulous conclusion of H.

The case of $\mathcal{E}_\mathcal{D}$ theories, where both the generator and the tester are used, is more involved. Consider the problem of deciding whether $Q = p$ `holds-at` T is a credulous conclusion of some $\mathcal{E}_\mathcal{D}$ theory. First the generator and tester are invoked as described above, restricting the search to stable models that contain Q. The outcome of this procedure is an initial stable model M that contains Q (which is minimal, wrt to the triples that fail, amongst the stable models that contain Q). Then, M is given as input to the tester which is asked to check whether there is a stable model better than M (i.e. with fewer failed triples) that includes the proposition $\neg p$ `holds-at` T. If the tester succeeds, then backtracking is required so that some other initial stable model M' that contains Q is possibly computed. If the tester fails, then Q is a credulous ($\mathcal{E}_\mathcal{D}$) conclusion. As mentioned above, for Q to be a skeptical ($\mathcal{E}_\mathcal{D}$) conclusion we also need to check that this process fails for $Q' = \neg p$ `holds-at` T.

5 Related Work and Conclusions

We have presented several encodings of subclasses of language $\mathcal{E}$ into answer set programming. It turns out that reasoning in language $\mathcal{E}$ spans different levels of the polynomial hierarchy, from polynomial time to Σ_2^P. The encodings that have been presented can be used directly to provide an implementation of language $\mathcal{E}$. More generally, they establish general links between answer set programming and languages for reasoning about actions and change that can encompass the main problems of frame, ramification and qualification.

The language $\mathcal{E}$ has limited expressive power, due to its propositional nature, in specifying static properties that refer to one situation or time point at a time. Despite this, the language is rich enough to express the dynamic properties required from a RAC formalism in order to encompass properly the main problems associated with reasoning about actions and change. As such, the language $\mathcal{E}$ can be taken as a representative example of RAC formalisms for the purposes of studying the complexity of temporal projection and how this varies as we address the different problems associated with this. We therefore believe that the results of this paper are general and relevant to most formalisms of RAC. Further work is needed to study this and how the proposed solutions within the ASP paradigm can be applied to other RAC frameworks.

Several other action description languages have been translated into ASP. Apart from the original but relatively restricted language $\mathcal{A}$, other descendants of this, such as the languages $\mathcal{C}$ [7], $\mathcal{AL}$ [2,1] and *Causal Logic* [6], have been recently translated into ASP. These translations, see e.g. [16,1], link the transition system semantics of these languages to the stable models or answer sets of the translated programs. Similarly, the language $\mathcal{K}$ is translated into ASP and implemented via the $\texttt{DLV}^{\mathcal{K}}$ system [4]. To the extend that the language $\mathcal{E}$ is related to all these frameworks our translation is similar to these translations. But these languages have a limited interest in the qualification problem approaching the problem of defeasible actions in a restricted way, via explicit abnormalities as compared to our approach of extending the semantics of the language and hence dealing with the qualification problem via, what we could call, implicit qualifications. To the best of our knowledge this is the first time that an action description language that integrates together all the three main problems of RAC has been translated to ASP together with the formal complexity results that underlie this translation. Our work provides a comparative (wrt the expressiveness of the language) and systematic investigation of the complexity of RAC and its relation to ASP and therefore could provide useful information on various computational issues within other frameworks.

As the language $\mathcal{E}$ is also implemented in terms of argumentation and abduction [9,13] our work also offers the opportunity to test and compare the computational behavior of the bottom up computational model of ASP with the top-down resolution based model of argumentation and abduction for problems of RAC. Other future work includes extending the ASP encodings to the case of $\mathcal{E}_{\mathcal{D}}$ theories with incomplete initial states or other extensions of language $\mathcal{E}$ not considered here, e.g. non-deterministic actions. Also a more complete study of the computational complexity of language $\mathcal{E}$ will possibly identify more classes

of theories where reasoning is easier than in the general case. An additional line of future work is extensive experimentation with the ASP encodings and comparison with different approaches such as `CCalc` and `DLV`.

References

1. M. Balduccini and M. Gelfond. Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3, 2003.
2. C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In J. Minker, editor, *Logic Based AI*. Kluwer, 2002.
3. T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System description: DLV. In *Proceedings of LPNMR-01*, 2001.
4. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The $DLV^{\mathcal{K}}$ system. *Artificial Intelligence*, 144, 2003.
5. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Logic Programming*, 17:301–322, 1993.
6. E. Giunchilgia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, to appear, 2003.
7. E. Giunchilgia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of AAAI-98*, 1988.
8. T. Janhunen, I. Niemela, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. In *Proceedings of KR'00*, 2000.
9. A. Kakas and L. Michael. Modeling complex domains of action and change. In *Proc. of NMR-02, Toulouse, France*, 2002.
10. A. Kakas and L. Michael. On the qualification problem and elaboration tolerance. In *AAAI Spring Symp. on Logical Formalization of Commonsense Reasoning*, 2003.
11. A. Kakas and R. Miller. Reasoning about actions, narratives and ramifications. *Electronic Transactions in Artificial Intelligence*, 1(4):39–72, 1997.
12. A. Kakas and R. Miller. A simple declarative language for describing narratives with ations. *Logic Programming*, 31:157–200, 1997.
13. A. Kakas, R. Miller, and F. Toni. An argumentation framework for reasoning about actions and change. In *Proc. of LPNMR-99*, 1999.
14. P. Liberatore. The complexity of the language A. *Electronic Transactions on Artificial Intelligence*, 1(1-3), 1997.
15. V. Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25 year perspective*, pages 357–373, 1999.
16. V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *Proc. of LPNMR'99*, 1999.
17. Y. Martin and M. Thielscher. Addressing the qualification problem in FLUX. In *Proc. of KI 2001*, pages 290–304, 2001.
18. M. Shanahan. An abductive event calculus planner. *Logic Programming*, 44, 2000.
19. T. Surjanen and I. Niemela. The Smodels system. In *Proceedings of LPNMR-01*, 2001.

Almost Definite Causal Theories

Semra Doğandağ[1], Paolo Ferraris[2], and Vladimir Lifschitz[2]

[1] Computer Engineering Department, Middle East Technical University, 06531 Ankara, Turkey
semra@ceng.metu.edu.tr
[2] Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712-1188, USA
{otto, vl}@cs.utexas.edu

Abstract. The language of nonmonotonic causal theories, defined by Norman McCain and Hudson Turner, is an important formalism for representing properties of actions. For causal theories of a special kind, called definite, a simple translation into the language of logic programs under the answer set semantics is available. In this paper we define a similar translation for causal theories of a more general form, called almost definite. Such theories can be used, for instance, to characterize the transitive closure of a binary relation. The new translation leads to an implementation of a subclass of almost definite causal theories that employs the answer set solver SMODELS as the search engine.

1 Introduction

The language of nonmonotonic causal theories defined in [McCain and Turner, 1997] is an important formalism for representing properties of actions. The postulates of a causal theory are *causal rules*–expressions of the form

$$G \Leftarrow F \tag{1}$$

where F and G are propositional formulas. Intuitively, rule (1) says that there is a cause for G to be true if F is true. For instance, the causal rule

$$p_{t+1} \Leftarrow a_t$$

can be used to describe the effect of an action a on a Boolean fluent p: if a is executed at time t then there is a cause for p to hold at time $t+1$.

Many useful causal theories contain rules of the form

$$l \Leftarrow l \tag{2}$$

where l is a literal ("if l is true then there is a cause for this"). In the semantics of causal theories, this rule expresses, intuitively, that l is true by default. For instance, the assumption that a Boolean fluent p is normally true can be

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 74–86, 2004.

expressed by the rules $p_t \Leftarrow p_t$. The frame problem [Shanahan, 1997] is solved in causal logic using rules similar to (2):

$$\begin{array}{l} p_{t+1} \Leftarrow p_{t+1} \wedge p_t \\ \neg p_{t+1} \Leftarrow \neg p_{t+1} \wedge \neg p_t. \end{array} \tag{3}$$

These rules express that fluent p obeys the commonsense law of inertia: if p holds at time t then it is assumed by default to hold at time $t+1$ also, and similarly for $\neg p$.

A causal theory is *definite* if the head G of each of its causal rules (1) is a literal or the 0-place connective $\bot$. A finite definite causal theory can be easily turned into an equivalent set of propositional formulas using the "literal completion" procedure defined in [McCain and Turner, 1997], and then queries about it can be answered by invoking a satisfiability solver. That translation is used in an implementation of the definite fragment of causal logic, called the Causal Calculator, or CCalc.[3] The Causal Calculator has been applied to several challenge problems in the theory of commonsense reasoning [Lifschitz *et al.*, 2000], [Lifschitz, 2000], [Akman *et al.*, 2003], [Campbell and Lifschitz, 2003].

An alternative computational procedure for answering questions about causal theories is to translate them into logic programs under the answer set semantics as in Proposition 6.7 from [McCain, 1997], and then invoke an answer set solver, such as SMODELS[4] [Doğandağ *et al.*, 2001]. McCain's translation, like literal completion, is limited to definite theories.

In this paper we extend McCain's translation to the class of "almost definite" causal theories, which includes all definite theories and covers some other interesting cases. One kind of interesting almost definite theories has to do with the idea of transitive closure, or reachability in a directed graph, which plays an important role in formal commonsense reasoning. We have used the new translation to extend the implementation of definite causal theories based on SMODELS to a class of almost definite causal theories.

After a review of causal theories and logic programs in the next two sections, we define the concept of an almost definite causal theory and the new translation in Section 4. Examples of almost definite theories that formalize commmonsense knowledge and their translations are discussed in Section 5, and the implementation in Section 6. In Section 7 we relate this paper to other work on causal logic and answer sets and discuss the computational complexity of almost definite causal theories. The proofs can be found in an extended version of this paper at `http://www.cs.utexas.edu/users/vl/papers/adct.ps` .

2 Causal Theories

As defined in the introduction, a *causal theory* is a set of *causal rules* of form (1), where F and G are propositional formulas.[5] These formulas are called the *body*

[3] `http://www.cs.utexas.edu/users/tag/ccalc/`.

[4] `http://www.tcs.hut.fi/Software/smodels/`.

[5] In [Giunchiglia *et al.*, 2003] the syntax of causal rules allows F and G to be "multi-valued propositional formulas." Causal theories in this more general sense can be

and the *head* of the rule. The semantics of causal theories [McCain and Turner, 1997] defines when an interpretation I of the underlying signature (that is, a function from atoms to truth values) is a model of a causal theory T, as follows. The *reduct* T^I of T relative to I is the set of the heads of the rules of T whose bodies are satisfied by I. We say that I is a *model* of T if I is the only model of T^I in the sense of propositional logic.

Take, for instance, the following causal theory T of the signature $\{p, q\}$:

$$\begin{array}{l} p \Leftarrow q \\ q \Leftarrow q \\ \neg q \Leftarrow \neg q. \end{array} \tag{4}$$

The interpretation I defined by $I(p) = I(q) = \mathbf{t}$ is a model of T. Indeed, I satisfies the bodies of the first two rules of T, so that the reduct T^I consists of the heads p, q of these rules; I is the only interpretation satisfying these formulas. The other 3 interpretations of $\{p, q\}$ are not models of T. Indeed, if $I(q) = \mathbf{f}$ then T^I is the set $\{\neg q\}$, which is satisfied by two interpretations; if $I(p) = \mathbf{f}$ and $I(q) = \mathbf{t}$ then the reduct is the set $\{p\}$, which is not satisfied by I.

The semantics of causal theories expresses the idea that a formula can be true only if the given causal rules guarantee that there is a cause for this. For instance, the rules in example (4) tell us that, under some conditions, there is a cause for p to be true, and similarly for q and $\neg q$, but these rules do not assert the existence of a cause for $\neg p$. Consequently, $\neg p$ cannot be true, which implies that p has to be true. Since the first rule of (4) is the only rule expressing the existence of a cause for p, the body q of that rule has to be true also. This informal reasoning explains why (4) has no models other than the interpretation that makes both p and q true.

Note that without the second rule, causal theory (4) would be inconsistent: q would have to be true, but there would have been no cause for this.

3 Logic Programs

The review of the answer set semantics in this section follows [Lifschitz *et al.*, 1999] The original definition of this semantics in [Gelfond and Lifschitz, 1988] and [Gelfond and Lifschitz, 1991] applied to programs of a more special form, without nested expressions.

Elementary expressions are literals and the symbols $\bot$ ("false") and $\top$ ("true"). *Nested expressions* are built from elementary expressions using the unary connective *not* (negation as failure) and the binary conncectives , (conjunction) and; (disjunction). A *logic program* is a set of *program rules* of the form

$$Head \leftarrow Body \tag{5}$$

where both *Head* and *Body* are nested expressions.

reduced to causal theories in the sense of [McCain and Turner, 1997] that are studied in this note.

The answer set semantics defines when a consistent set X of literals is an answer set for a program Π. As a preliminary step, we define when X *satisfies* a nested expression F (symbolically, $X \models F$), as follows:

- $X \models l$ if $l \in X$ for any literal l,
- $X \models \top$,
- $X \not\models \bot$,
- $X \models F, G$ if $X \models F$ and $X \models G$,
- $X \models F; G$ if $X \models F$ or $X \models G$,
- $X \models not\ F$ if $X \not\models F$.

We say that X satisfies a program Π (symbolically, $X \models \Pi$), if for each rule (5) in Π, if $X \models Body$ then $X \models Head$.

The *reduct* Π^X of Π relative to X is obtained from Π by substituting each outermost expression of the form $not\ F$ in Π with $\bot$ if $X \models F$, and with $\top$ otherwise. We say that X is an *answer set* for Π if X is a minimal set satisfying Π^X.

For instance, it is easy to check that the program

$$\begin{array}{l} p \leftarrow not\ \neg q \\ q \leftarrow not\ \neg q \\ \neg q \leftarrow not\ q \end{array} \tag{6}$$

has two answer sets:

$$\{p, q\},\ \{\neg q\}. \tag{7}$$

In this paper, we will sometimes identify the logic program connectives , and ; with the symbols used in propositional formulas $\wedge$ and $\vee$. Under this convention, nested expressions that do not contain negation as failure are identical to propositional formulas that are formed from literals using the connectives $\wedge$, $\vee$, $\top$ and $\bot$. Such propositional formulas are said to be *in standard form*, and the literals they are formed from will be called their *component literals*.

4 Translation

Recall that in a definite causal theory, the head of every rule is a literal or $\bot$.[6] In this section, T stands for a causal theory whose rules have the form

$$G \supset H \Leftarrow F. \tag{8}$$

where F, G and H are in standard form. When G is $\top$, we will identify the head $G \supset H$ of this rule with H, so that definite rules (with the body written in standard form) can be viewed as a special case of (8).

[6] The definition of a definite causal theory in [Giunchiglia *et al.*, 2003, Section 2.6] includes also a finiteness assumption. That assumption is essential for the concept of completion discussed in that paper, but it is not needed for our present purposes.

Let $\bar{l}$ stand for the literal complementary to l. We say that l is *default false* if T contains the rule

$$\bar{l} \Leftarrow \bar{l} \tag{9}$$

We say that T is *almost definite* if, in each of its rules (8),

- the component literals of G are default false, and
- the component literals of H are default false, or H is a conjunction of literals.

Definite theories are almost definite: in each of their rules, G is $\top$, and H is a literal or $\bot$. The pair of rules

$$\begin{array}{l} \neg q \supset p \Leftarrow \top \\ q \Leftarrow q \end{array} \tag{10}$$

is an example of an almost definite theory that is not definite.

Now we will describe a translation that turns any almost definite causal theory T into a logic program Π_T. For any standard formula F, by F_{not} we denote the nested expression obtained from F by replacing each component literal l with $not\ \bar{l}$. For instance,

$$(\neg p \wedge q)_{not} = not\ p, not\ \neg q.$$

For any almost definite causal theory T, the logic program Π_T is defined as the set of the program rules

$$H \leftarrow G, F_{not} \tag{11}$$

for all causal rules (8) in T.

For instance, the translation of (4) is the program

$$\begin{array}{l} p \leftarrow \top, not\ \neg q \\ q \leftarrow \top, not\ \neg q \\ \neg q \leftarrow \top, not\ q \end{array}$$

which is equivalent to (6). (On equivalent transformations of logic programs, see [Lifschitz *et al.*, 1999, Section 4]). The translation of (10) can be similarly written as

$$\begin{array}{l} p \leftarrow \neg q \\ q \leftarrow not\ \neg q. \end{array} \tag{12}$$

The theorem below expresses the soundness of this translation. We identify each interpretation with the set of literals that are satisfied by it. Clearly this set is complete: for each atom a, it contains either a or $\neg a$.

Theorem 1. *An interpretation is a model of an almost definite causal theory T iff it is an answer set for Π_T.*

An alternative way of proving this theorem is mentioned in [Lee, 2003].

Note that this theorem does not say anything about the answer sets for the translation Π_T that are not complete. For instance, the first of the answer sets (7)

for program (6) is complete, and the second is not; in accordance with Theorem 1, the first answer set is identical to the only model of the corresponding causal theory (4). The only answer set for the translation (12) of causal theory (10) is $\{q\}$; it is incomplete, and accordingly (10) has no models.[7]

Rules of Π_T can be more complex than allowed by the preprocessor LPARSE of the system SMODELS. (The syntax of LPARSE does not permit disjunctions in the bodies of rules, as well as conjunctions and non-exclusive disjunctions in the heads of rules.) If, however, the formulas F and G in a rule (8) are conjunctions of literals, and H is a literal or $\bot$, then the translation (11) of that rule can be processed by LPARSE. These conditions are satisfied in many interesting cases, including the examples of almost definite causal theories discussed in the next section.

5 Examples

5.1 Transitive Closure

The transitive closure of a binary relation P on a set A can be described by an almost definite causal theory as follows. For any $x, y \in A$ such that xPy, the theory includes the causal rule

$$p(x, y) \Leftarrow \top. \tag{13}$$

In the remaining causal rules, x, y and z stand for arbitrary elements of A. By default, P does not hold:

$$\neg p(x, y) \Leftarrow \neg p(x, y). \tag{14}$$

If xPy then there is a cause for x and y to satisfy the transitive closure of P:

$$tc(x, y) \Leftarrow p(x, y), \tag{15}$$

and there is a cause for the implication $tc(y, z) \supset tc(x, z)$ to hold:

$$tc(y, z) \supset tc(x, z) \Leftarrow p(x, y). \tag{16}$$

Finally, by default, the transitive closure relation does not hold:

$$\neg tc(x, y) \Leftarrow \neg tc(x, y). \tag{17}$$

The following theorem expresses that this representation of transitive closure in causal logic is adequate. By P^* we denote the transitive closure of P.

Theorem 2. *Causal theory (13)–(17) has a unique model. In this model M, an atom is true iff it has the form $p(x, y)$ where xPy, or the form $tc(x, y)$ where xP^*y.*

[7] The incomplete answer sets of a logic program can be eliminated by adding the constraints $\leftarrow not\ a, not\ \neg a$ for all atoms a.

$$
\begin{aligned}
p(x,y) &\leftarrow \top && \text{if } xPy\\
\neg p(x,y) &\leftarrow \mathit{not}\ p(x,y)\\
tc(x,y) &\leftarrow \mathit{not}\ \neg p(x,y)\\
tc(x,z) &\leftarrow tc(y,z), \mathit{not}\ \neg p(x,y)\\
\neg tc(x,y) &\leftarrow \mathit{not}\ tc(x,y)
\end{aligned}
$$

Fig. 1. Translation of causal theory (13)–(17).

An alternative way of proving this theorem is shown in [Lee, 2003].

If we attempt to make the almost definite causal theory (13)–(17) definite by replacing (16) with the definite rule

$$tc(x,z) \Leftarrow p(x,y) \wedge tc(y,z)$$

then the assertion of Theorem 2 will become incorrect [Giunchiglia *et al.*, 2003, Section 7.2].

The logic program corresponding to causal theory (13)–(17) is shown in Figure 1.

Rules (15) and (16) above can be replaced with

$$p(x,y) \supset tc(x,y) \Leftarrow \top$$

and

$$p(x,y) \wedge tc(y,z) \supset tc(x,z) \Leftarrow \top.$$

The assertion of Theorem 2 holds for the modified theory also. Since the atoms $p(x,y)$ are default false, the modified theory is almost definite, so that Theorem 1 can be used to turn it into a logic program. Its translation differs from the one shown in Figure 1 in that it does not have the combination $\mathit{not}\ \neg$ in the third and forth lines.

The fact that the atoms $p(x,y)$ are assumed to be defined by rules of the forms (13) and (14) is not essential for the validity of Theorem 2, or for the claim that the theory is almost definite; the relation P can be characterized by any definite causal theory with a unique model. But the modification described above would not be almost definite in the absence of rules (14).

Transitive closure often arises in work on formalizing commonsense reasoning. For instance, Erik Sandewall's description of the Zoo World[8] says about the neighbor relation among positions that it "is symmetric, of course, and the transitive closure of the neighbor relation reaches all positions." Because of the difficulty with expressing transitive closure in the definite fragment of causal logic, this part of the specification of the Zoo World is disregarded in the paper by Akman *et al.* [2003] where the Zoo World is formalized in the input of language of CCalc.

Here is another example of this kind. The apartment where Robby the Robot lives consists of several rooms connected by doors, and Robby is capable of

[8] `http://www.ida.liu.se/ext/etai/lmw/` .

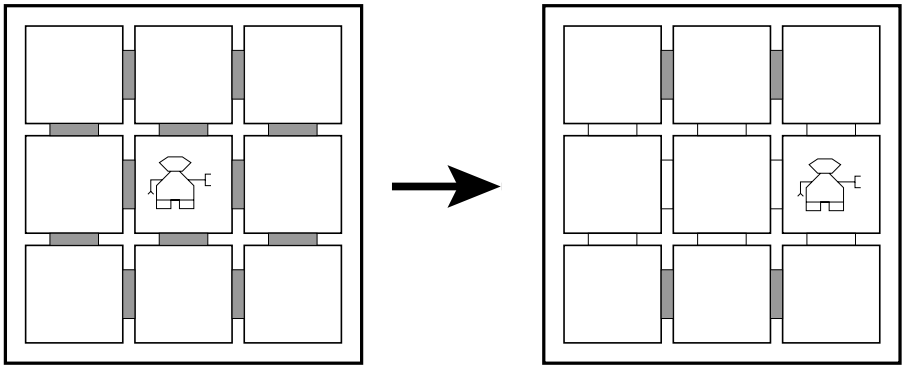


Fig. 2. Robby's apartment is a 3×3 grid, with a door between every pair of adjacent rooms. Initially Robby is in the middle, and all doors are locked. The goal of making every room accessible from every other can be achieved by unlocking 8 doors, and the robot will have to move to other rooms in the process.

moving around and of locking and unlocking the doors. This is a typical action domain of the kind that are easily described by definite causal theories. But the assignment given to Robby today is to unlock enough doors to make any room accessible from any other (Figure 2). To express this goal in the language of causal logic we need, for any time instant t, the transitive closure of the relation "there is currently an unlocked door connecting Room i with Room j." In the spirit of the representation discussed above, the transitive closure can be defined by the causal rules

$$\begin{aligned} \mathit{Accessible}(i,j)_t &\Leftarrow \mathit{Unlocked}(i,j)_t \\ \mathit{Accessible}(j,k)_t \supset \mathit{Accessible}(i,k)_t &\Leftarrow \mathit{Unlocked}(i,j)_t \\ \neg\mathit{Accessible}(i,j)_t &\Leftarrow \neg\mathit{Accessible}(i,j.)_t \end{aligned} \tag{18}$$

5.2 Two Gears

This domain, invented by Marc Denecker, is described in [McCain, 1997, Section 7.5.5] as follows:

> Imagine that there are two gears, each powered by a separate motor. There are switches that toggle the motors on and off, and a button that moves the gears so as to connect or disconnect them from one another. The motors turn the gears in opposite (i.e., compatible) directions. A gear is caused to turn if either its motor is on or it is connected to a gear that is turning.

McCain's representation of this domain as a causal theory is shown in Figure 3. The first 4 lines describe the direct effects of actions. The next 4 lines have the form (3) and express the commonsense law of inertia. The 8 lines that

$$
\begin{aligned}
\neg MotorOn(G(i))_{t+1} &\Leftarrow Toggle(S(i))_t \wedge MotorOn(G(i))_t \\
MotorOn(G(i))_{t+1} &\Leftarrow Toggle(S(i))_t \wedge \neg MotorOn(G(i))_t \\
\neg Connected_{t+1} &\Leftarrow Push_t \wedge Connected_t \\
Connected_{t+1} &\Leftarrow Push_t \wedge \neg Connected_t \\
MotorOn(G(i))_{t+1} &\Leftarrow MotorOn(G(i))_{t+1} \wedge MotorOn(G(i))_t \\
\neg MotorOn(G(i))_{t+1} &\Leftarrow \neg MotorOn(G(i))_{t+1} \wedge \neg MotorOn(G(i))_t \\
Connected_{t+1} &\Leftarrow Connected_{t+1} \wedge Connected_t \\
\neg Connected_{t+1} &\Leftarrow \neg Connected_{t+1} \wedge \neg Connected_t \\
MotorOn(G(i))_0 &\Leftarrow MotorOn(G(i))_0 \\
\neg MotorOn(G(i))_0 &\Leftarrow \neg MotorOn(G(i))_0 \\
Connected_0 &\Leftarrow Connected_0 \\
\neg Connected_0 &\Leftarrow \neg Connected_0 \\
Toggle(S(i))_t &\Leftarrow Toggle(S(i))_t \\
\neg Toggle(S(i))_t &\Leftarrow \neg Toggle(S(i))_t \\
Push_t &\Leftarrow Push_t \\
\neg Push_t &\Leftarrow \neg Push_t
\end{aligned}
$$

$(i = 1, 2;\ t = 0, \ldots, n-1);$

$$
\begin{aligned}
Turning(G(i))_t &\Leftarrow MotorOn(G(i))_t \\
Turning(G(1))_t \equiv Turning(G(2))_t &\Leftarrow Connected_t \\
\neg Turning(G(i))_t &\Leftarrow \neg Turning(G(i))_t
\end{aligned}
$$

$(i = 1, 2;\ t = 0, \ldots, n).$

Fig. 3. Two gears domain.

follow say that the initial values of fluents and the execution of actions are "exogenous." The last 3 lines express that a gear's motor being on causes it to turn, that the gears being connected causes them to turn (and not turn) together, and that by default the gears are assumed not to turn.

Because of the second line from the end, this theory is not definite. But we can make it almost definite by replacing that line with

$$
\begin{aligned}
Turning(G(1))_t \supset Turning(G(2))_t &\Leftarrow Connected_t \\
Turning(G(2))_t \supset Turning(G(1))_t &\Leftarrow Connected_t.
\end{aligned}
$$

By Proposition 4(i) from [Giunchiglia *et al.*, 2003], this transformation does not change the set of models.

The corresponding logic program is shown in Figure 4. Many occurrences of the combination *not* $\neg$ in this program can be dropped without changing the answer sets.

6 Implementation

The system for answering queries about definite causal theories described in [Doğandağ *et al.*, 2001] has the same input language as CCALC but uses a different computational mechanism: it turns the given theory into a logic program and invokes SMODELS to find the program's answer sets.

$$\begin{aligned}
\neg MotorOn(G(i))_{t+1} &\leftarrow not\ \neg Toggle(S(i))_t, not\ \neg MotorOn(G(i))_t \\
MotorOn(G(i))_{t+1} &\leftarrow not\ \neg Toggle(S(i))_t, not\ MotorOn(G(i))_t \\
Connected_{t+1} &\leftarrow not\ \neg Push_t, not\ Connected_t \\
\neg Connected_{t+1} &\leftarrow not\ \neg Push_t, not\ \neg Connected_t \\
MotorOn(G(i))_{t+1} &\leftarrow not\ \neg MotorOn(G(i))_{t+1}, not\ \neg MotorOn(G(i))_t \\
\neg MotorOn(G(i))_{t+1} &\leftarrow not\ MotorOn(G(i))_{t+1}, not\ MotorOn(G(i))_t \\
Connected_{t+1} &\leftarrow not\ \neg Connected_{t+1}, not\ \neg Connected_t \\
\neg Connected_{t+1} &\leftarrow not\ Connected_{t+1}, not\ Connected_t \\
MotorOn(G(i))_0 &\leftarrow not\ \neg MotorOn(G(i))_0 \\
\neg MotorOn(G(i))_0 &\leftarrow not\ MotorOn(G(i))_0 \\
Connected_0 &\leftarrow not\ \neg Connected_0 \\
\neg Connected_0 &\leftarrow not\ Connected_0 \\
Toggle(S(i))_t &\leftarrow not\ \neg Toggle(S(i))_t \\
\neg Toggle(S(i))_t &\leftarrow not\ Toggle(S(i))_t \\
Push_t &\leftarrow not\ \neg Push_t \\
\neg Push_t &\leftarrow not\ Push_t
\end{aligned}$$

$(i = 1, 2;\ t = 0, \ldots, n-1)$,

$$\begin{aligned}
Turning(G(i))_t &\leftarrow not\ \neg MotorOn(G(i))_t \\
Turning(G(2))_t &\leftarrow Turning(G(1))_t, not\ \neg Connected_t \\
Turning(G(1))_t &\leftarrow Turning(G(2))_t, not\ \neg Connected_t \\
\neg Turning(G(i))_t &\leftarrow not\ Turning(G(i))_t
\end{aligned}$$

$(i = 1, 2;\ t = 0, \ldots, n)$.

Fig. 4. Translation of the two gears domain.

We have extended this system to a class of theories that are almost definite (or can be made almost definite by simple equivalent transformations), for which the logic program obtained by the translation from Section 4 is nondisjunctive.[9]

At the beginning of the translation process, the system finds all rules of the form $l \Leftarrow l$ in the given causal theory, and thus identifies all default false literals. (To be more precise, the input consists of *schematic rules*, or rules with variables. The system finds all schematic rules of the form $l \Leftarrow l$ and adds the ground instances of $\bar{l}$ to the list of default false literals.) The head of each rule that contains more than one atom is converted to conjunctive normal form, and then the rule is replaced by a group of rules whose heads are disjunctions of literals. Then the systems attempts to rewrite each of these disjunctions in the form $G \supset H$ where G is a conjunction of default false literals and H is a literal or $\bot$. (A similar transformation was applied to an equivalence in Section 5.2 above.) If this can be done for the head of every rule then the corresponding

[9] The need for this restriction is related to the fact that SMODELS is not applicable to general disjunctive programs. Using the answer set solver DLV (`http://www.dbai.tuwien.ac.at/proj/dlv/`) instead of SMODELS would be a way to overcome this limitation.

logic program Π_T is formed, the rules from Footnote ([7]) are added to it, and SMODELS is called to solve the given problem.

To test the system, we have used the examples from Section 5 expressed in the "action language" notation [Giunchiglia and Lifschitz, 1998], [Giunchiglia *et al.*, 2003, Section 4] that eliminates the need to mention time explicitly. For instance, the assumption that executing the action $Go(i)$ makes the fluent $At(i)$ true can be expressed in this notation by the "causal law"

$$Go(i) \textbf{ causes } At(i)$$

that is viewed as an abbreviation for

$$At(i)_{t+1} \Leftarrow Go(i)_t.$$

Similarly, the causal law

$$\textbf{nonexecutable } Go(i) \textbf{ if } At(j) \wedge \neg Unlocked(i,j)$$

(the robot cannot go to Room i if there is no unlocked door connecting Room i with the room where the robot is) stands for

$$\bot \Leftarrow Go(i)_t \wedge At(j)_t \wedge \neg Unlocked(i,j)_t.$$

Causal laws like these are used to describe the effects of actions—going to other rooms and unlocking doors—and restrictions on their executability. In addition, the formalization contains the definition of accessibility by causal laws that are expanded into causal rules (18):

$$\begin{array}{l}\textbf{caused } Accessible(i,j) \textbf{ if } Unlocked(i,j)\\ \textbf{caused } Accessible(j,k) \supset Accessible(i,k) \textbf{ if } Unlocked(i,j)\\ \textbf{default } \neg Accessible(i,j).\end{array}$$

The solution shown in Figure 2 corresponds to one of the plans generated by this system. The plan consist of 11 steps: 3 *Go* actions and 8 *Unlock* actions.

7 Conclusion

The main theorem of this paper extends Proposition 6.7 from [McCain, 1997] from definite to almost definite causal theories. Another special case of the main theorem that was known before is the case when

- every atom in the language of T is default false, that is to say, T contains the causal rule $\neg a \Leftarrow \neg a$ for every atom a, and
- in every other rule (8) of T, F is a conjunction of negative literals, G is a conjunction of atoms, and H is a disjunction of atoms.

This special case is covered essentially by the lemma from [Giunchiglia *et al.*, 2003, Section 7.3]. What is interesting about this case is that by forming the translation Π_T of a causal theory T satisfying the conditions above we can get an *arbitrary* set of rules of the form

$$a_1; \ldots; a_m \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_p \tag{19}$$

where $a_1, \ldots, a_m, b_1, \ldots, b_n, c_1, \ldots, c_p$ are atoms, plus the "closed world assumption" rules

$$\neg a \leftarrow not\ a$$

for all atoms a. Since the problem of existence of an answer set for a finite set of rules of the form (19) is Σ_2^P-hard [Eiter and Gottlob, 1993, Corollary 3.8], it follows that the problem of existence of a model for an almost definite causal theory is Σ_2^P-hard also. This fact shows that from the complexity point of view almost definite causal theories are as general as arbitrary causal theories.

On the other hand, if the formula H in every rule (8) of an almost definite causal theory T is a literal or $\bot$ then the corresponding logic program Π_T is nondisjunctive. Consequently, the problem of existence of a model for the almost definite causal theories satisfying this condition is in class NP, just as for definite causal theories. This condition is satisfied, for instance, for both examples discussed in Section 5.

If a causal theory T is definite then the corresponding logic program Π_T is tight in the sense of [Erdem and Lifschitz, 2003]. The answer sets for a finite tight program can be computed by eliminating classical negation from it in favor of additional atoms and then generating the models of the program's completion [Babovich *et al.*, 2000]. This process is essentially identical to the use of literal completion mentioned in the introduction. If T is almost definite but not definite then the program Π_T, generally, is not tight.

An earlier version of CCalc had a mechanism for answering queries about arbitrary causal theories, not necessarily definite or even almost definite, by calling a satisfiability solver several times. That mechanism was based on an algorithm that is, in principle, complete, but very inefficient.

Our future plans include extending the implementation discussed in Section 6 to causal theories with multi-valued formulas defined in [Giunchiglia *et al.*, 2003].

Acknowledgements. We are grateful to Joohyung Lee and Hudson Turner for comments on a draft of this paper. This work was partially supported by the Texas Higher Education Coordinating Board under Grant 003658-0322-2001, by the National Science Foundation under Grant INT-0004433, and by the Scientific and Technical Research Council of Turkey under Grant 101E024.

References

[Akman *et al.*, 2003] Varol Akman, Selim Erdoğan, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Representing the Zoo World and the Traffic World in the language of the Causal Calculator. *Artificial Intelligence*, 2003. To appear.

[Babovich *et al.*, 2000] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming.[10] In *Proc. Eighth Int'l Workshop on Non-Monotonic Reasoning*, 2000.

[Campbell and Lifschitz, 2003] Jonathan Campbell and Vladimir Lifschitz. Reinforcing a claim in commonsense reasoning.[11] In *Working Notes of the AAAI Spring*

[10] `http://arxiv.org/abs/cs.ai/0003042.`

[11] `http://www.cs.utexas.edu/users/vl/papers/sams.ps.`

Symposium on Logical Formalizations of Commonsense Reasoning, 2003.

[Doğandağ *et al.*, 2001] Semra Doğandağ, Ferda N. Alpaslan, and Varol Akman. Using stable model semantics (SMODELS) in the Causal Calculator (CCALC). In *Proc. 10th Turkish Symposium on Artificial Intelligence and Neural Networks*, pages 312–321, 2001.

[Eiter and Gottlob, 1993] Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In Dale Miller, editor, *Proc. ILPS-93*, pages 266–278, 1993.

[Erdem and Lifschitz, 2003] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Giunchiglia and Lifschitz, 1998] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. AAAI-98*, pages 623–630. AAAI Press, 1998.

[Giunchiglia *et al.*, 2003] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories.[12] *Artificial Intelligence*, 2003. To appear.

[Lee, 2003] Joohyung Lee. Nondefinite vs. definite causal theories. In this volume.

[Lifschitz *et al.*, 1999] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[Lifschitz *et al.*, 2000] Vladimir Lifschitz, Norman McCain, Emilio Remolina, and Armando Tacchella. Getting to the airport: The oldest planning problem in AI. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 147–165. Kluwer, 2000.

[Lifschitz, 2000] Vladimir Lifschitz. Missionaries and cannibals in the Causal Calculator. In *Principles of Knowledge Representation and Reasoning: Proc. Seventh Int'l Conf.*, pages 85–96, 2000.

[McCain and Turner, 1997] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.

[McCain, 1997] Norman McCain. *Causality in Commonsense Reasoning about Actions.*[13] PhD thesis, University of Texas at Austin, 1997.

[Shanahan, 1997] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.

[12] `http://www.cs.utexas.edu/users/vl/papers/nmct.ps.`

[13] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.Z.`

Simplifying Logic Programs under Uniform and Strong Equivalence*

Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran

Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,michael,tompits,stefan}@kr.tuwien.ac.at

Abstract. We consider the simplification of logic programs under the stable-model semantics, with respect to the notions of strong and uniform equivalence between logic programs, respectively. Both notions have recently been considered for nonmonotonic logic programs (the latter dates back to the 1980s, though) and provide semantic foundations for optimizing programs with input. Extending previous work, we investigate syntactic and semantic rules for program transformation, based on proper notions of consequence. We furthermore provide encodings of these notions in answer-set programming, and give characterizations of programs which are semantically equivalent to positive and Horn programs, respectively. Finally, we investigate the complexity of program simplification and determining semantical equivalence, showing that the problems range between coNP and Π_2^P complexity, and we present some tractable cases.

1 Introduction

Implementations of answer-set solvers such as DLV [4], Smodels [21], or ASSAT [13] led to the consideration of practical applications of nonmonotonic logic programs in the last years, but also renewed interest in the study of foundational properties. In particular, semantical notions of equivalence between logic programs such as *strong equivalence* have been studied (cf. [11,22,23,17,12,3]): Programs P_1 and P_2 are strongly equivalent, if, for any set R of rules, the programs $P_1 \cup R$ and $P_2 \cup R$ are equivalent under the stable semantics, i.e., have the same set of stable models. This can be used to simplify a logic program P [23,16]: if a subprogram Q of P is strongly equivalent to a (simpler) program Q', then we can replace Q by Q'.

Since strong equivalence is rather strict, the more liberal notion of *uniform equivalence* [20,14] has been considered in [5,18], where R is restricted to sets of *facts*. A hierarchical component C within a program P may be replaced by a uniformly equivalent set of rules C', providing the global hierarchical component structure of P is not affected (which is a simple syntactic check). As recently discussed by Pearce and Valverde [18], uniform and strong equivalence are essentially the only concepts of equivalence obtained by varying the logical form of the program extensions.

* This work was partially supported by the Austrian Science Fund (FWF) under project Z29-N04, and the European Commission under projects FET-2001-37004 WASP and IST-2001-33570 INFOMIX.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 87–99, 2004.

This paper continues and extends the work in [5], which focused on semantical characterizations of uniform equivalence for disjunctive logic programs (DLPs). More specifically, we consider here the issue of simplifying DLPs under uniform and strong equivalence under different aspects. Our main contributions are briefly summarized as follows:

(1) We consider a method using *local transformation rules* to simplify programs, where we deal both with syntactic and semantic transformation rules. Syntactic transformation rules for strong equivalence have previously been considered by Osorio et al. [16]. Besides rules from there and from [2], we also examine the recent notion of *s-implication* [24], as well as a new transformation rule for head-cycle free rules, called *local shifting*. Both preserve uniform equivalence, and the former also strong equivalence. The semantic transformation rules employ logical consequence and remove redundant rules and literals. The method interleaves syntactic and semantic transformation rules, respecting that the former have much lower (polynomial) complexity compared to the intractability of the semantic rules.

(2) We provide answer-set programming (ASP) solutions for checking whether $P \models_s r$ resp. $P \models_u r$, where $\models_s$ denotes the consequence operator under strong-equivalence models (SE-models) [22,23] and $\models_u$ refers to consequence under uniform-equivalence models (UE-models) [5]. Note that deciding $P \models_u r$ is Π_2^P-complete [5], and hence the full power of DLPs is needed to decide this problem, whilst deciding $P \models_s r$ is "only" coNP-complete. We remark that Pearce and Valverde [18] have provided a tableau system for deciding uniform equivalence, and that Janhunen and Oikarinen [9,10] provided an alternative method in terms of ASP for testing strong and ordinary equivalence between normal logic programs.

(3) Beyond local transformations, we present general conditions under which programs possess equivalent programs belonging to syntactic subclasses of DLPs (which have special algorithms for ASP computation). In particular, we provide semantical characterizations (in terms of conditions on models) of programs which are uniformly resp. strongly equivalent to programs which are positive or Horn. Furthermore, we give similar conditions in terms of classical consequence for strong equivalence.

(4) We analyze the computational complexity of the problems encountered. In particular, we consider the cost of the semantic simplification rules, and the cost of deciding whether a given DLP is equivalent to a syntactically simpler program. While most of these problems are unsurprisingly intractable (coNP-complete, and in few cases for uniform equivalence Π_2^P-complete), we obtain a polynomial-time result for deciding whether a normal logic program without constraints of form $\leftarrow A_1, \ldots, A_n$ is strongly equivalent to some positive DLP or to some Horn program, respectively.

Given the availability of efficient answer-set solvers, such as the systems mentioned previously, enabling automated simplifications of logic programs has become an important issue. This is even more the case since with the development of applications using ASP solvers, an ever growing number of programs are automatically generated, leaving the burden of optimizations to the underlying ASP system. Our results might well be used for *offline optimizations* of application programs in ASP solvers.

For space reasons, proofs are omitted; they are given in an extended version of this paper.

2 Preliminaries

We deal with propositional disjunctive logic programs, containing rules r of form

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n,$$

$n \geq m \geq l \geq 0$, where all a_i are atoms from a finite set of propositional atoms, At, and not denotes default negation. The *head* of r is the set $H(r) = \{a_1, \ldots, a_l\}$, and the *body* of r is the set $B(r) = \{a_{l+1}, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$. We also define $B^+(r) = \{a_{l+1}, \ldots, a_m\}$ and $B^-(r) = \{a_{m+1}, \ldots, a_n\}$. Moreover, for a set of atoms $A = \{a_1, \ldots, a_n\}$, $not\ A$ denotes the set $\{not\ a_1, \ldots, not\ a_n\}$.

A rule r is *normal*, if $l \leq 1$; *definite*, if $l = 1$; *positive*, if $n = m$; and *Horn*, if it is normal and positive. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*; and if moreover $B^-(r) = \emptyset$, then r is a *positive constraint*. If $B(r) = \emptyset$, r is a *fact*, written as $a_1 \vee \cdots \vee a_l$ if $l > 0$, and as $\bot$ otherwise.

A *disjunctive logic program* (DLP), P, is a finite set of rules. P is called a *normal logic program* (NLP) (resp., a *definite*, *positive*, *Horn* program), if every rule in P is normal (resp., definite, positive, Horn). We also define $P^+ = \{r \in P \mid B^-(r) = \emptyset\}$.

In a DLP P, an atom a *positively depends* on an atom b if $a \in H(r)$ for some rule $r \in P$, and either $b \in B^+(r)$, or some $c \in B^+(r)$ positively depends on b; $r \in P$ is *head-cycle free* (HCF) *in* P if no distinct atoms $a, b \in H(r)$ mutually positively depend on each other. A DLP P is HCF [1] if each $r \in P$ is HCF in P.

We recall the stable-model semantics for DLPs [8,19]. Let I be an interpretation, i.e., a subset of At. Then, an atom a is *true under* I, symbolically $I \models a$, iff $a \in I$, and *false under* I otherwise. For a rule r, $I \models H(r)$ iff some $a \in H(r)$ is true under I, and $I \models B(r)$ iff (i) each $a \in B^+(r)$ is true under I, and (ii) each $a \in B^-(p)$ is false under I. I *satisfies* r, denoted $I \models r$, iff $I \models H(r)$ whenever $I \models B(r)$. Furthermore, I is a *model* of a program P, denoted $I \models P$, iff $I \models r$, for all $r \in P$. As usual, $P \models r$ iff $I \models r$, for each model I of P.

The *Gelfond-Lifschitz reduct* of a program P *relative to* a set of atoms I is the positive program $P^I = \{H(r) \leftarrow B^+(r) \mid r \in P,\ B^-(r) \cap I = \emptyset\}$. An interpretation I is a *stable model* of a program P iff I is a minimal model (under set inclusion) of P^I. The set of all stable models of P is denoted by $\mathcal{SM}(P)$.

Several notions of equivalence between logic programs have been considered in the literature (cf., e.g., [11,14,20]). Under stable semantics, two DLPs P and Q are regarded as equivalent, denoted $P \equiv Q$, iff $\mathcal{SM}(P) = \mathcal{SM}(Q)$. The more restrictive forms of *strong equivalence* and *uniform equivalence* are as follows:

Definition 1. *Let P and Q be two DLPs. Then,*

(*i*) *P and Q are* strongly equivalent, *or* s-equivalent, *denoted $P \equiv_s Q$, iff, for any set R of rules, the programs $P \cup R$ and $Q \cup R$ are equivalent, i.e., $P \cup R \equiv Q \cup R$.*

(*ii*) *P and Q are* uniformly equivalent, *or* u-equivalent, *denoted $P \equiv_u Q$, iff, for any set F of non-disjunctive facts, $P \cup F$ and $Q \cup F$ are equivalent, i.e., $P \cup F \equiv Q \cup F$.*

Obviously, $P \equiv_s Q$ implies $P \equiv_u Q$ but not vice versa (see Example 1 below). Both notions of equivalence, however, enjoy interesting semantical characterizations. As shown in [11], strong equivalence is closely related to the non-classical logic of here-and-there, which was adapted to logic-programming terms by Turner [22,23]:

Definition 2. *Let P be a DLP, and let $X, Y \subseteq At$ such that $X \subseteq Y$. The pair (X, Y) is an* SE-model *of P (over At), if $Y \models P$ and $X \models P^Y$. By $M_s^{At}(P)$ we denote the set of all SE-models over At of P.*

In what follows, we usually leave the set At implicit and write $M_s(P)$ *simpliciter* instead of $M_s^{At}(P)$.

Proposition 1 ([22,23]). *For any DLP P and Q, $P \equiv_s Q$ iff $M_s(P) = M_s(Q)$.*

Recently, the following pendant to SE-models, characterizing uniform equivalence for (finite) logic programs, has been defined [5].

Definition 3. *Let P be a DLP and $(X, Y) \in M_s(P)$. Then, (X, Y) is an* UE-model *of P iff, for every $(X', Y) \in M_s(P)$, it holds that $X \subset X'$ implies $X' = Y$. By $M_u(P)$ we denote the set of all UE-models of P.*

Proposition 2 ([5]; cf. also [18]). *For any DLP P and Q, $P \equiv_u Q$ iff $M_u(P) = M_u(Q)$.*

Example 1. Consider $P = \{a \leftarrow\}$ and $Q = \{a \leftarrow not\, b;\ a \leftarrow b\}$ over $At = \{a, b\}$. As easily checked, we have $M_s(P) = \{(\{a, b\}, \{a, b\}), (\{a\}, \{a, b\}), (\{a\}, \{a\})\}$ and $M_s(Q) = M_s(P) \cup \{(\emptyset, \{a, b\})\}$. Thus, $P \not\equiv_s Q$. However, $(\emptyset, \{a, b\}) \notin M_u(Q)$. In fact, it holds that $M_u(P) = M_s(P)$ and $M_u(Q) = M_u(P)$. Therefore, $P \equiv_u Q$.

Finally, we define consequence relations associated to SE- and UE-models in the usual manner.

Definition 4. *Let P be a DLP, r a rule, and $e \in \{s, u\}$. Then, $P \models_e r$ iff, for each $(X, Y) \in M_e(P)$, $(X, Y) \models_e r$, i.e., $Y \models r$ and $X \models \{r\}^Y$. Furthermore, we write $P \models_e Q$ iff $P \models_e r$, for every $r \in Q$.*

Proposition 3. *For any DLP P and Q, $P \equiv_e Q$ iff $P \models_e Q$ and $Q \models_e P$, $e \in \{s, u\}$.*

3 Local Program Transformations

Given a logic program P and a notion of equivalence $\equiv_e$ ($e \in \{s, u\}$), we aim at a procedure to systematically simplify P to a program P' which is e-equivalent to P. As a starting point, we consider *local modifications*, i.e., iterative modifications of P on a rule-by-rule basis. Later on, in Section 5, we present results for replacing—under certain conditions—the *entire program* by a simpler, e-equivalent program.

Figure 1 gives the general structure of such a simplification algorithm, using different kinds of simplifications: A first simplification can be obtained by analyzing the syntax of the program and by applying appropriate syntactic transformation rules. After this "static" analysis, the notion of consequence induced by the given type of equivalence can be used for further semantic simplifications by means of consequence tests (see also next section). Systematic consequence testing can be applied to simplify rules by identifying redundant literals for removal, as well as for program simplification by identifying redundant rules.

Algorithm $Simplify(P, \equiv_e)$

Input: A DLP P and a notion of equivalence $\equiv_e$.
Output: A simplified DLP P' such that $P' \equiv_e P$.

var P' : DLP, I : changeInfo ;
$P' := P$; $I := all$;
while $I \neq \emptyset$ **do**
$P' := Simplify\_Syntactic(P', \equiv_e, I)$;
$P' := Remove\_Redundant\_Rule(P', \equiv_e, I)$;
if $(I = \emptyset)$ **then** $P' := Remove\_Redundant\_Literal(P', \equiv_e, I)$; **fi**
od
return P';

Fig. 1. Generic simplification algorithm.

The algorithm depicted in Figure 1 interleaves syntactic with semantic simplifications, giving preference to syntactic ones since they are easier to accomplish. Furthermore, it uses a data structure for recording latest changes (changeInfo I) in order to restrict the set of applicable transformation rules as well as to restrict their range of application: By means of this information, in each iteration, only a particular set of syntactic transformation rules has to be tested for a specific rule. Of course, this basic algorithm can be further optimized to fit particular needs. Finally, we remark that $Simplify$ is sound and that it is an anytime algorithm, since we use rewriting rules for a transformation-based calculus, where $P = P_0$ is transformed by applying rules T_i iteratively: $P = P_0 \xrightarrow{T_{i_1}} P_1 \xrightarrow{T_{i_2}} P_2 \cdots P_{k-1} \xrightarrow{T_{i_k}} P_k$, such that $P_j \equiv_e P_{j+1}$, $0 \leq j < k$.

3.1 Syntactic Transformation Rules

Basic transformation rules. A set of basic syntactic transformation rules has been introduced and initially studied by Brass and Dix [2]. Table 1 briefly summarizes these rules, called *elimination of tautologies* (TAUT), *positive-* and *negative reduction* (RED$^+$ and RED$^-$, respectively), *partial evaluation* (GPPE), *weak partial evaluation* (WGPPE), *elimination of non-minimal rules* (NONMIN), and *elimination of contradictions* (CONTRA).

Osorio et al. [16] reported results for TAUT, RED$^+$, RED$^-$, GPPE, and NONMIN under strong equivalence. The positive results for TAUT, RED$^-$, and NONMIN directly carry over to uniform equivalence. The programs $P = \{a \leftarrow not\ b\}$ and $Q = \{a \leftarrow b;\ b \leftarrow c\}$ show that RED$^+$ and GPPE also fail to preserve uniform equivalence. Moreover, it is easily verified that CONTRA preserves both notions of equivalence.

While GPPE does neither preserve s-equivalence nor u-equivalence, both are preserved, however, by the weak variant WGPPE.

Proposition 4. *Let P be a DLP and consider $r_1, r_2 \in P$ such that $a \in B^+(r_1)$ and $a \in H(r_2)$, for some atom a. Then, $P \cup \{r'\} \equiv_s P$, where r' is given by $H(r_1) \cup (H(r_2) \setminus \{a\}) \leftarrow (B^+(r_1) \setminus \{a\}) \cup not\ B^-(r_1) \cup B(r_2)$.*

Table 1. Syntactic transformation rules.

Name	Condition	Transformation
TAUT	$H(r) \cap B^+(r) \neq \emptyset$	$P' = P \setminus \{r\}$
RED$^+$	$a \in B^-(r_1)$, $\nexists r_2 \in P : \ a \in H(r_2)$	$P' = P \setminus \{r_1\} \cup \{r'\}^\dagger$
RED$^-$	$H(r_2) \subseteq B^-(r_1)$, $B(r_2) = \emptyset$	$P' = P \setminus \{r_1\}$
NONMIN	$H(r_2) \subseteq H(r_1)$, $B(r_2) \subseteq B(r_1)$	$P' = P \setminus \{r_1\}$
GPPE	$a \in B^+(r_1)$, $G_a \neq \emptyset$, for $G_a = \{r_2 \in P \mid a \in H(r_2)\}$	$P' = P \setminus \{r_1\} \cup G'^\ddagger_a$
WGPPE	same condition as for GPPE	$P' = P \cup G'^\ddagger_a$
CONTRA	$B^+(r) \cap B^-(r) \neq \emptyset$	$P' = P \setminus \{r\}$
S-IMP	$r, r' \in P$, $r \lhd r'$	$P' = P \setminus \{r'\}$
LSH	$r \in P$, $\lvert H(r)\rvert > 1$, r head-cycle free in P	$P' = P \setminus \{r\} \cup r^\rightarrow$

† $r' : H(r_1) \ \leftarrow \ B^+(r_1) \cup not\,(B^-(r_1) \setminus \{a\})$.
‡ $G'_a = \{H(r_1) \cup (H(r_2) \setminus \{a\}) \ \leftarrow \ (B^+(r_1) \setminus \{a\}) \cup not\, B^-(r_1) \cup B(r_2) \mid r_2 \in G_a\}$.

Note that WGPPE creates new rules and may make other transformations applicable. Furthermore, it may be applied exponentially often during iteration steps, creating further (even minimal) rules. Hence, to avoid an exponential explosion in the simplification algorithm, the number of applications of WGPPE may be polynomially bounded.

S-implication (*S-IMP*). Recently, Wang and Zhou [24] introduced the notion of s-implication:

Definition 5. *A rule r' is an* s-implication *of a rule $r \neq r'$, symbolically $r \lhd r'$, iff there exists a set $A \subseteq B^-(r')$ such that* (*i*) $H(r) \subseteq H(r') \cup A$, (*ii*) $B^-(r) \subseteq B^-(r') \setminus A$, *and* (*iii*) $B^+(r) \subseteq B^+(r')$.

For example, if $r = a \vee b$ and $r' = a \leftarrow not\, b$, then $r \lhd r'$ (choose $A = \{b\}$).

Proposition 5. *Let P be a DLP, and $r, r' \in P$ such that $r \lhd r'$. Then, $P \setminus \{r'\} \equiv_s P$.*

Example 2. Consider the program $P = \{r_1 : a \leftarrow not\, b;\ r_2 : a \vee b \leftarrow\}$. Since $r_2 \lhd r_1$, P can be simplified to the strongly equivalent program $P' = \{a \vee b \leftarrow\}$.

Local shifting (*LSH*). Finally, we present a new transformation rule, called *local shifting*. It relies on the concept of head-cycle freeness and turns out to preserve uniform equivalence, but not strong equivalence.

For any rule r, define $r^\rightarrow = \{a \leftarrow B(r), not\,(H(r) \setminus \{a\}) \mid a \in H(r)\}$ if $H(r) \neq \emptyset$, and $r^\rightarrow = \{r\}$ otherwise. Furthermore, for any DLP P, let $P^\rightarrow = \bigcup_{r \in P} r^\rightarrow$.

Definition 6. *The rule* local shifting (*LSH*) *allows replacing a rule r in a DLP P by $r^\rightarrow$, providing r is head-cycle free in P.*

Theorem 1. *Let P be a DLP and $r \in P$ HCF in P. Then, $P \equiv_u (P \setminus \{r\}) \cup r^\rightarrow$.*

Example 3. Consider $P = \{a \vee b \leftarrow;\ c \vee d \leftarrow b;\ c \leftarrow a, d;\ d \leftarrow b, c\}$. Here, $a \vee b \leftarrow$ is HCF in P, whilst $c \vee d \leftarrow b$ is not. Hence, under $\equiv_u$, program P can be simplified to $P' = (P \setminus \{a \vee b \leftarrow\}) \cup \{a \leftarrow not\, b; b \leftarrow not\, a\}$.

Table 2 summarizes the results for syntactic transformation rules under both uniform and strong equivalence.

Table 2. Syntactic transformations preserving e-equivalence ($e \in \{s, u\}$).

Eq.	TAUT	RED$^+$	RED$^-$	NONMIN	GPPE	WGPPE	CONTRA	S-IMP	LSH
$\equiv_s$	yes*	no*	yes*	yes*	no*	yes	yes	yes	no
$\equiv_u$	yes	no	yes	yes	no	yes	yes	yes	yes

*Results due to Osorio et al. [16].

3.2 Semantic Transformation Rules

More powerful than the above syntactic transformation rules for program simplification are semantic ones based on $\models_e$, in the light of Propositions 1–3. However, checking the applicability of these rules is intractable in general, while checking the applicability of the above syntactic rules is tractable.

Brass and Dix [2] considered *supraclassicality* (SUPRA): If $P \models a$, then $P \equiv P \cup \{a \leftarrow\}$. However, SUPRA fails under $\equiv_u$ (and thus also under under $\equiv_s$). A simple counterexample is $P = \{a \leftarrow not\, b;\ a \leftarrow b;\ a \leftarrow b, c;\ b \leftarrow a, c\}$. Clearly, $P \models a$, but P has a UE-model $(\{c\}, \{a, b, c\})$ which is not a UE-model of $P \cup \{a \leftarrow\}$.

Rule simplification. One way is minimizing individual rules r, based on property NONMIN: replace r by a subsumer r' (with $H(r') \subseteq H(r)$ and $B(r') \subseteq B(r)$) such that $(P \setminus \{r\}) \cup \{r'\} \equiv_e P$. Rule simplification (or *condensation*) thus amounts to the systematic search for such r', which must satisfy $P \models_e r'$. The easiest way is literalwise removal, i.e., where r' is r with some literal in r removed.

Rule redundancy. In addition, we can simplify P by removing redundant rules. Proposition 3 implies that $r \in P$ is redundant in P iff $P \setminus \{r\} \models_e r$.

Notice that rule simplification might create redundant rules, as shown by the following example, which is well-known in the context of computing a minimal cover of a set of functional dependencies in databases.

Example 4. Consider the program $P = \{c \leftarrow a, b;\ a \leftarrow b, c;\ b \leftarrow a;\ b \leftarrow c\}$. None of the rules in P is redundant, neither under strong nor under uniform consequence. However, b can be removed in both $c \leftarrow a, b$ and $a \leftarrow b, c$, yielding $P' = \{c \leftarrow a;\ a \leftarrow c;\ b \leftarrow a;\ b \leftarrow c\}$. Here, $b \leftarrow c$ or $b \leftarrow a$ can be eliminated.

We remark that some of the syntactic transformation rules, such as TAUT, RED$^-$, NONMIN, and CONTRA, can be regarded as special cases of semantic transformation rules. However, others such as LSH, cannot be viewed this way.

We may bypass e-consequence and stay with relation $\models$ in some cases. This is true, e.g., for positive programs, which is an easy corollary of the next result, which generalizes an analogous result of [5] for uniform equivalence to e-equivalence.

Theorem 2. *Let P be a DLP and $e \in \{s, u\}$. Then, the following conditions are equivalent:*

(*i*) *$P \models_e r$ iff $P \models r$, for every rule r.*
(*ii*) *For every $(X, Y) \in M_e(P)$, it holds that $X \models P$.*

Corollary 1. *For any positive DLP P and any rule r, $P \models_e r$ iff $P \models r$, $e \in \{s, u\}$.*

Thus, for every positive program, a UE- and SE-canonical form is given by any irredundant prime CNF, written in positive rule form. Hence, minimization of any program which is semantically equivalent to a positive one is possible in polynomial time with an NP-oracle; detecting this property is considered in Section 5.

3.3 Computational Cost of $Simplify$ and Incomplete Simplification

As pointed out above, testing the applicability and execution of each syntactic rule in Section 3.1 valid for $\equiv_e$ is clearly polynomial. Since all transformation rules except WGPPE shrink or remove program rules, $Simplify(P, \equiv_e)$ is polynomial with an oracle for $\equiv_e$ if WGPPE is applied at most polynomially often.

As discussed in Section 6, following from Corollary 1, deciding rule and literal redundancy is intractable in general. Interestingly, the following extension of Corollary 1 yields sound inference in terms of $\models$ for arbitrary DLPs.

Proposition 6. *For any DLP P and rule r, $P^+ \models r$ implies $P \models_e r$, for $e \in \{s, u\}$.*

However, the condition is not necessary; e.g., for $P = \{a \leftarrow b;\ a \leftarrow not\, b\}$ and $r = a \leftarrow not\, b$, we have $P \models_s r$ but $P^+ \not\models r$. Note that $P \models r$ for positive r does not imply $P \models_e r$, as shown by $P = \{a \leftarrow not\, a\}$ and $r = a \leftarrow$.

By exploiting Proposition 6, we can simplify any DLP P using its positive part P^+, without guaranteeing completeness. Notice that if P^+ is Horn (e.g., if P is a NLP), this sound redundancy test is polynomial, and is in coNP for general P^+. Thus, for P^+ being Horn it is cheaper than the sound and complete test for NLPs (which is coNP-complete for both $\models_s$ and $\models_u$), and for arbitrary P^+ for $\models_u$ (which is Π_2^P-complete; cf. Section 6).

4 Consequence Testing

In order to enable the systematic tests from the previous section by means of answer-set solvers, we provide here efficient translations of checking SE- and UE-consequence into (disjunctive) logic programs under the stable-model semantics.

We employ the following notation. For any set I of atoms, we define $I' = \{v' \mid v \in I\}$ and $\bar{I} = \{\bar{v} \mid v \in I\}$, where v' and $\bar{v}$ are globally new atoms, for each atom v. Furthermore, r' (resp., $\bar{r}$) denotes the result of replacing in r each occurrence of an atom v by v' (resp., $\bar{v}$). Informally, primed atoms will be used to characterize X in a UE- or SE-model (X, Y); whilst atoms of form $\bar{v}$ will be used to encode the negation of v.

4.1 Encoding SE-Consequence

Based on results in [12,17,23], a SAT encoding of $\models_s$ is easily obtained. In what follows, we describe an analogous encoding in terms of normal logic programs.

Definition 7. *For any DLP P and atom set $V = \{v_1, \ldots, v_n\}$, define $S_{P,V}$ as:*

$$
\begin{array}{lll}
(1) & v_i \leftarrow not\ \bar{v}_i; \quad \bar{v}_i \leftarrow not\ v_i, & 1 \leq i \leq n, \\
(2) & v'_i \leftarrow v_i, not\ \bar{v}'_i; \quad \bar{v}'_i \leftarrow not\ v'_i, & 1 \leq i \leq n, \\
(3) & \leftarrow B^+(r), not\ B^-(r), not\ H(r), & r \in P, \\
(4) & \leftarrow B^+(r'), not\ B^-(r), not\ H(r'), & r \in P.
\end{array}
$$

Intuitively, (1) and (2) guess sets $X, Y \subseteq V$ such that $X \subseteq Y$, where atoms in X are represented by primed variables; (3) checks $Y \models P$; and (4) $X \models P^Y$.

The next result describes the desired encoding of $\models_s$.

Theorem 3. *Let P, Q be DLPs, V the set of atoms occurring in $P \cup Q$, and w a fresh atom. Then, $P \models_s Q$ iff the following NLP has no stable model:*

$$
\begin{array}{l}
S_{P,V} \cup \{\leftarrow not\ w\} \cup \{w \leftarrow B^+(r), not\ B^-(r), not\ H(r); \\
\qquad\qquad w \leftarrow B^+(r'), not\ B^-(r), not\ H(r') \mid r \in Q\}.
\end{array}
$$

4.2 Encoding UE-Consequence

Definition 8. *Given a DLP P, a set of atoms $V = \{v_1, \ldots, v_n\}$, and sets of new atoms $V_i = \{v_{i,1}, \ldots, v_{i,n}\}$, for $i \in \{1, \ldots, n\}$, $U_{P,V}$ is defined as follows:*

$$
\begin{array}{lll}
(1) & S_{P,V}, & \\
(2) & v_{i,j} \leftarrow v_i, \bar{v}'_i, v'_j; \quad v_{i,i} \leftarrow v_i, \bar{v}'_i, & 1 \leq i, j \leq n, \\
(3) & \leftarrow v_i, \bar{v}'_i, v_j, \bar{v}'_j, not\ v_{i,j}; \quad \leftarrow v_i, \bar{v}'_i, \bar{v}_j, v_{i,j}, & 1 \leq i, j \leq n, \\
(4) & H(r_i) \leftarrow v_i, \bar{v}'_i, B^+(r_i), not\ B^-(r), & r \in P, 1 \leq i \leq n,
\end{array}
$$

where rule r_i is the result of replacing in r each atom v_j by $v_{i,j}$.

Intuitively, $U_{P,V}$ works as follows. First, $S_{P,V}$ yields all SE-models (X, Y) of P. Then, (2)–(4) check if no Z with $X \subset Z \subset Y$ satisfies $Z \models P^Y$. This is done as follows: For each $v_i \in Y \setminus X$, (2) initializes sets $X_i = X \cup \{v_i\}$ via atoms V_i. Then, (4) yields, for each $v_i \in Y \setminus X$, minimal sets $Z_i \supseteq X_i$ satisfying $Z_i \models P^Y$. Finally, constraints (3) apply for each such $Z_i \neq Y$. Hence, we get a stable model iff, for each i such that $v_i \in Y \setminus X$, the minimal set Z_i matches Y. But then, $(X, Y) \in M_u(P)$. Formally, we have the following result:

Lemma 1. *Let P be a DLP over atoms $V = \{v_1, \ldots, v_n\}$. Then, $\mathcal{SM}(U_{P,V})$ is given by $\{X' \cup (\bar{V}' \setminus \bar{X}') \cup Y \cup (\bar{V} \setminus \bar{Y}) \cup J_{X,Y} \mid X, Y \subseteq V, (X, Y) \in M_u(P)\}$, where $J_{X,Y} = \{v_{i,j} \mid v_i \in Y \setminus X, v_j \in Y\}$.*

Theorem 4. *Let P,Q be DLPs, V the set of atoms in $P \cup Q$, and u, w, s new atoms. Then, $P \models_u Q$ iff the program $C^U_{P,Q}$, given as follows, has no stable model:*

$$
\begin{array}{lll}
(1) & U_{P,V}, & \\
(2) & u \leftarrow B^+(r), not\ B^-(r), not\ H(r), & r \in Q, \\
(3) & u \leftarrow B^+(r'), not\ B^-(r), not\ H(r'), & r \in Q, \\
(4) & v''_i \leftarrow v'_i; \quad v''_i \leftarrow v_i, not\ \bar{v}''_i; \quad \bar{v}''_i \leftarrow not\ v''_i, & 1 \leq i \leq n, \\
(5) & w \leftarrow not\ v''_i, v_i; \quad s \leftarrow v''_i, not\ v'_i, & 1 \leq i \leq n, \\
(6) & \leftarrow not\ w, not\ u; \quad \leftarrow not\ s, not\ u, & \\
(7) & \leftarrow B^+(r''), not\ B^-(r), not\ H(r''), not\ u, & r \in Q.
\end{array}
$$

The intuition behind $C^U_{P,Q}$ is as follows. (1) computes all UE-models (X, Y) of P via atoms V' and V (characterizing X' resp. Y). We must check that each of them is also a UE-model of Q, and if so, the program should have no stable model. Therefore, (2) checks $Y \models Q$ and (3) $X \models Q^Y$. If this fails, $(X, Y) \notin M_s(Q)$, and thus $(X, Y) \notin M_u(Q)$. Whenever this is detected, we derive u, which is used to disable the constraints (6)–(7). Finally, we must check whether no Z exists such that $X \subset Z \subset Y$ and $(Z, Y) \in M_s(Q)$. To this end, (4) guesses a set Z such that $X \subseteq Z \subseteq Y$, and (5) derives w if some $p \in Y \setminus Z$ exists, and s if some $q \in Z \setminus X$ exists. Using (6), we remain with those Z which are in between X and Y. Finally, (7) checks $(Z, Y) \notin M_s(Q)$.

Now let I be any stable model of $C^U_{P,Q}$. If $u \in I$, then the members of V,V' in I give an UE-model of P which is not an SE-model of Q; and if $u \notin I$, then V, V'' give an SE-model (Z, Y) of Q where $(X, Y) \in M_u(P)$ with $X \subset Z$.

Notice that both $U_{P,V}$ and $C^U_{P,Q}$ are normal (resp., HCF) if P is normal (resp., HCF). Hence, the test for $P \models_u Q$ yields a normal (resp., HCF) program if P is normal (resp., HCF), which is in coNP compared to Π^P_2-completeness in general [5]. $U_{P,V}$ is thus appealing from this point of view. However, the size of $U_{P,V}$ is quadratic in the size of P. In the extended paper, we give a linear-size program replacing $U_{P,V}$, but it is neither normal nor HCF regardless of P.

5 Global Simplifications

In this section, we consider the following issue: Given a DLP P from some class $\mathcal{C}$, under what conditions is there a program Q from a class $\mathcal{C}'$ such that $P \equiv_e Q$? This question is crucial for program transformation and simplification from a more general perspective than local transformations. We succeed here providing characterizations where $\mathcal{C}'$ is the class of positive or Horn programs.

Theorem 5. *Let P be a DLP. Then, there exists a positive program Q such that $P \equiv_e Q$ iff, for all $(X, Y) \in M_e(P)$, it holds that $X \models P$, where $e \in \{s, u\}$.*

Theorem 6. *For any DLP P, let $M^1_e(P) = \{X \mid (X, Y) \in M_e(P)\}$, for $e \in \{s, u\}$. Then, there exists some Horn program Q such that $P \equiv_e Q$ iff (i) $M^1_e(P) \models P$, and (ii) $M^1_e(P)$ is closed under intersection.*

Note that Theorem 5 implies that if P is e-equivalent to some positive program Q, then we can obtain Q by shifting not -literals to the head. We can exploit this observation for a simple test. Let, for any rule r, r^{ls} be the rule such that $H(r^{ls}) = H(r) \cup B^-(r)$, and $B(r^{ls}) = B^+(r)$ (i.e., the "left shift" of r).

Theorem 7. *Let P be any DLP and $e \in \{s, u\}$. Then, there exists a positive program Q such that $P \equiv_e Q$ iff $P \models_e r^{ls}$, for every $r \in P$ such that $B^-(r) \neq \emptyset$.*

To test for the existence of an e-equivalent Horn program Q, we just need to add a check whether the models of P are intersection closed (which is in coNP).

In the case of $\models_s$, we can even get rid of SE-consequence and come up with a test using $\models$ only. In the following, we characterize strong equivalence of any program P,

Table 3. Complexity of deciding rule and literal redundancy.

$\equiv_s$ / $\equiv_u$	Horn LP	Normal LP	Positive LP	HCF LP	DLP
$r \in P$ redundant?	P	coNP	coNP	coNP	coNP/Π_2^P
ℓ in $r \in P$ redundant?	P	coNP	coNP	coNP	coNP/Π_2^P

possessing a classical model, to some positive program Q in terms of $\models$; if P has no classical model, then $P \equiv_s \{\bot\}$. Note that $\{\bot\}$ is positive.

Recall that $(X, Y) \in M_e$ requires $X \subseteq Y$, but the reduct P^Y of program P may have models X' such that $X' \not\subseteq Y$. To select exactly those X' such that $X' \subseteq Y$, let, for any DLP P and interpretation I, $P^I_{\leq}$ be the positive logic program $P^I \cup \{ \leftarrow a \mid I \not\models a\}$.

Proposition 7. *Let P be a DLP having a classical model, and let r be a positive rule. Then, $P \models_s r$ iff $P^M_{\leq} \models r$, for every maximal model M of P.*

Note that the proposition fails for non-positive r. Indeed, consider the program $P = \{a \leftarrow b;\ a \leftarrow not\, b\}$ and $r = a \leftarrow not\, b$. The unique maximal model of P is $At = \{a, b\}$, and $P^{At}_{\leq} = \{a \leftarrow b\}$, but $P^{At}_{\leq} \not\models a \leftarrow not\, b$.

Theorem 8. *Let P, Q be DLPs, where P is positive having a classical model. Then, $P \equiv_s Q$ iff* (i) *$P \models Q$, and* (ii) *$Q^M_{\leq} \models P$, for every maximal model M of P.*

We finally arrive at the desired characterization of semantic equivalence to some positive program, by exploiting that P^+ is a singular candidate for Q.

Theorem 9. *Let P be any DLP without positive constraints. Then, there exists some positive program Q such that $P \equiv_s Q$ iff $P^+ \models P \setminus P^+$.*

For strong equivalence of P to some Horn theory Q, we have in the right-hand side the additional condition that the models of P^+ are closed under intersection.

6 Complexity

In this section, we address complexity issues related to the problems above. Recall that we deal with propositional programs only.

The complexity of deciding redundant literals and rules is given in Table 3, for various classes of programs, where entries represent completeness for the respective complexity classes. The upper complexity bounds are easily obtained from the complexity of deciding $P \models_s r$ resp. $P \models_u r$ for the respective program class $\mathcal{C}$ in Table 3 (cf. [5]), since $P \in \mathcal{C}$ implies $P \setminus \{r\} \in \mathcal{C}$ in each case. The hardness parts for positive programs, HCF programs, and DLPs under $\equiv_s$ are easily obtained from the coNP-completeness of deciding $P \models v$ for an atom v and a positive HCF P. For Horn programs, the problem is polynomial since $P \models_s r$ and $P \models_u r$ coincide with $P \models r$, which is polynomial for Horn P. The Π_2^P-hardness of DLPs under $\equiv_u$ can be shown by suitable adaptions of proofs in [5]. However, redundancy checking is tractable in certain cases:

Table 4. Complexity of deciding if for $P \in \mathcal{C}$ (row) some $Q \in \mathcal{C}'$ (column) exists with $P \equiv_e Q$.

$\equiv_s$ / $\equiv_u$	Positive LP	Horn LP	constraint-free positive LP	Definite Horn
Normal LP	coNP	coNP	P/coNP	P/coNP
HCF LP	coNP	coNP	coNP	coNP
Disjunctive LP	coNP/Π_2^P	coNP/in Π_2^P	coNP/Π_2^P	coNP/in Π_2^P

Theorem 10. *Let P be a DLP without positive constraints such that P^+ is normal. Then, checking redundancy of a positive rule $r \in P$ under $\equiv_s$ is decidable in polynomial time, and likewise checking redundancy of a literal ℓ in a positive rule $r \in P$.*

We next consider the complexity of deciding whether a given program is equivalent to some program belonging to a specific syntactic subclass of DLPs. The generic problem is as follows:

Instance: A program P (possibly from a restricted class of programs, $\mathcal{C}$).
Question: Does there exist some program Q from a fixed class of programs, $\mathcal{C}'$, such that $P \equiv_e Q$, where $e \in \{s, u\}$ is fixed?

Our results concerning this task are summarized in Table 4, where entries stand for completeness for the respective complexity classes unless stated otherwise. Notice that definite Horn programs are constraint-free Horn programs.

Unsurprisingly, the problem is intractable in general. The upper bounds for the non-polynomial cases can be easily derived from the results in the previous sections and the complexity of $\models_s$, which is coNP-complete for general DLPs (as follows from [22, 23]), and of $\models_u$, which is Π_2^P-complete in general but coNP-complete for HCF (and, in particular, normal) logic programs.

Note that the rows for NLPs and HCF programs coincide under $\equiv_u$, since $P \equiv_u P^{\rightarrow}$ holds for each HCF program P, and the shifting program $P^{\rightarrow}$ is constraint-free iff P is constraint-free. However, under $\equiv_s$, we have a different picture, and we can find some interesting tractable cases from Theorem 9:

Theorem 11. *Let P be a positive-constraint free DLP such that P^+ is normal. Then, deciding if there is some positive program Q such that $P \equiv_s Q$ is feasible in polynomial time, and likewise deciding whether there is some Horn program Q with $P \equiv_s Q$.*

7 Conclusion

We have pushed further the work on nonmonotonic logic-program simplification under the notions of strong and uniform equivalence (cf. [11,22,23,20,5,16]) by providing syntactic and semantic rules for program transformation and giving characterizations of programs which are semantically equivalent to positive and Horn programs. Similar characterizations for equivalence to *normal* programs are considered in a companion paper [6]. Future work concerns implementing the methods presented, as well as extending the results to programs with two kinds of negation and to the case of programs with variables. The current results provide a solid basis, though, for lifting them using common techniques.

References

1. R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
2. S. Brass and J. Dix. Semantics of (Disjunctive) Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, 40(1):1–46, 1999.
3. D. J. de Jongh and L. Hendriks. Characterizations of Strongly Equivalent Logic Programs in Intermediate Logics. *Theory and Practice of Logic Programming*, 3(3):259–270, 2003.
4. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer, 2000.
5. T. Eiter and M. Fink. Uniform Equivalence of Logic Programs under the Stable Model Semantics. In *Proc. ICLP-03*. To appear.
6. T. Eiter, M. Fink, H. Tompits, and S. Woltran. Eliminating Disjunction from Propositional Logic Programs under Stable Model Preservation. In *Proc. ASP-03*. Available at: http://CEUR-WS.org/Vol-78/, 2003.
7. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22:364–418, 1997.
8. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
9. T. Janhunen and E. Oikarinen. Testing the Equivalence of Logic Programs under Stable Model Semantics. In *Proc. JELIA-02*, LNCS 2424, pp. 493–504. Springer, 2002.
10. T. Janhunen and E. Oikarinen. LPEQ and DLPEQ – Translators for Automated Equivalence Testing of Logic Programs. In *Proc. LPNMR-03*. To appear.
11. V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
12. F. Lin. Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic. In *Proc. KR-02*, pp. 170–176. Morgan Kaufmann, 2002.
13. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proc. AAAI-2002*, pp. 112–117. AAAI Press / MIT Press, 2002.
14. M. J. Maher. Equivalences of Logic Programs. In Minker [15], pp. 627–658.
15. J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufman, Washington DC, 1988.
16. M. Osorio, J. Navarro, and J. Arrazola. Equivalence in Answer Set Programming. In *Proc. LOPSTR 2001*, LNCS 2372, pp. 57–75. Springer, 2001.
17. D. Pearce, H. Tompits, and S. Woltran. Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In *Proc. EPIA 2001*, LNCS 2258, pp. 306–320. Springer, 2001.
18. D. Pearce and A. Valverde. Some Types of Equivalence for Logic Programs and Equilibrium Logic. In *Proc. APPIA-GULP-PRODE 2003*, 2003.
19. T. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.
20. Y. Sagiv. Optimizing Datalog Programs. In Minker [15], pp. 659–698.
21. P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, 2002.
22. H. Turner. Strong Equivalence for Logic Programs and Default Theories (Made Easy). In *Proc. LPNMR-01*, LNCS 2173, pp. 81–92. Springer, 2001.
23. H. Turner. Strong Equivalence Made Easy: Nested Expressions and Weight Constraints. *Theory and Practice of Logic Programming*, 3(4–5):609–622, 2003.
24. K. Wang and L. Zhou. Comparisons and Computation of Well-founded Semantics for Disjunctive Logic Programs. *ACM Transactions on Computational Logic*. To appear. Available at: http://arxiv.org/abs/cs.AI/0301010, 2003.

Towards Automated Integration of Guess and Check Programs in Answer Set Programming *

Thomas Eiter[1] and Axel Polleres[2]

[1] Institut für Informationssysteme, TU Wien, A-1040 Wien, Austria
eiter@kr.tuwien.ac.at
[2] Institut für Informatik, Universität Innsbruck, A-6020 Innsbruck, Austria
axel.polleres@uibk.ac.at

Abstract. Many NP-complete problems can be encoded in the answer set semantics of logic programs in a very concise way, where the encoding reflects the typical "guess and check" nature of NP problems: The property is encoded in a way such that polynomial size certificates for it correspond to stable models of a program. However, the problem-solving capacity of full disjunctive logic programs (DLPs) is beyond NP at the second level of the polynomial hierarchy. While problems there also have a "guess and check" structure, an encoding in a DLP is often non-obvious, in particular if the "check" itself is co-NP-complete; usually, such problems are solved by interleaving separate "guess" and "check" programs, where the check is expressed by inconsistency of the check program. We present general transformations of head-cycle free (extended) logic programs into stratified disjunctive logic programs which enable one to integrate such "guess" and "check" programs automatically into a single disjunctive logic program. Our results complement recent results on meta-interpretation in ASP, and extend methods and techniques for a declarative "guess and check" problem solving paradigm through ASP.

1 Introduction

Answer set programming (ASP) [19,7] is widely proposed as a useful tool for expressing properties in NP, where solutions and polynomial time proofs for such properties correspond to answer sets of normal logic programs, which cover by well-known complexity results the class NP. An example for such a property is whether a given graph has a legal 3-coloring, where any such coloring is itself a certificate for this property.

However, we also might encounter situations in which we want to express a problem which is complementary to some NP problem, and thus belongs to the class co-NP; it is widely believed that in general, not all such problems are in NP and hence not always a polynomial-size certificate checkable in polynomial time exists. One such problem is, e.g., the property that a graph is *not* 3-colorable. Such properties p can analogously be expressed by a normal logic program (equivalently, by a head-cycle free disjunctive logic program [1]) Π_p, where the property holds iff Π_p has no answer set at all.

* The major part of this work has been conducted at TU Wien, supported by FWF (Austrian Science Funds) projects P14781 and Z29-N04 and European Commission grants FET-2001-37004 WASP and IST-2001-33570 INFOMIX.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 100–113, 2004.

Checks in co-NP typically occur as subproblems within more complex problems which have complexity higher than NP, for instance:

Quantified Boolean Formulas (QBFs): Evaluating a QBF, where we have to check, given a QBF of the form $\exists X \forall Y \Phi(X, Y)$, and an assignment σ to the variables X, whether $\forall Y \Phi(\sigma(X), Y)$ evaluates to true.

Strategic Companies: Checking whether a set of companies is strategic (cf. [11]).

Conformant Planning: Checking whether a given plan is conformant [8], provided executability of actions is polynomially decidable (cf. [4,22]).

Further examples can be found in [6,5]. In general, the corresponding logic program Π_p for this check can be easily formulated and the overall problem (evaluating the QBF, finding a strategic companies set resp. a conformant plan) solved in a 2-step approach:

1. Generate a candidate solution by means of a logic program Π_{guess}.
2. Check the solution by another logic program Π_{check} (=Π_p).

However, it is often not clear how to combine Π_{guess} and Π_{check} into a *single* program Π_{solve} which solves the overall problem. Simply taking the union $\Pi_{guess} \cup \Pi_{check}$ does not work, and rewriting is needed. Theoretical results [6] informally give strong evidence that for problems with Σ_2^P-complexity, it is required that Π_{check} (given as a normal logic program or a head-cycle free disjunctive logic program) is rewritten into a disjunctive logic program Π'_{check} such that the answer sets of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$ yield the solutions of the problem, where Π'_{check} emulates the inconsistency check for Π'_{check} as a minimal model check, which is co-NP-complete for disjunctive programs. This becomes even more complicated by the fact that Π'_{check} must not crucially rely on the use of negation, since it is essentially determined by the Π_{guess} part. These difficulties can make rewriting Π_{check} to Π'_{check} a formidable and challenging task.

In this paper, we present a generic method for rewriting Π_{check} automatically by using a meta-interpreter approach. In particular, we make the following contributions:

(1) We provide a polynomial-time transformation $tr(\Pi)$ from propositional head-cycle-free [1] (extended) disjunctive logic programs (HDLPs) Π to disjunctive logic programs (DLPs), such that the following conditions hold:

- **T1** Each answer set S' of $tr(\Pi)$ corresponds to an answer set S of Π, such that $S = \{l \mid \texttt{inS}(l) \in S'\}$ for some predicate $\texttt{inS}(\cdot)$.
- **T2** If the original program has no answer sets, then $tr(\Pi)$ has exactly one designated answer set Ω, which is easily recognizable.
- **T3** The transformation is of the form $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$, where $F(\Pi)$ is a factual representation of Π and Π_{meta} is a fixed *meta-interpreter*.
- **T4** $tr(\Pi)$ is *modular* (at the syntactic level), i.e., $tr(\Pi) = \bigcup_{r \in \Pi} tr(r)$. Moreover, it is a stratified DLP [20,21] and uses negation only in its "deterministic" part.

We also describe optimizations and a transformation to positive DLPs, and show that in a precise sense, modular transformations to such programs do not exist.

(2) We show how to use $tr(\cdot)$ for integrating separate guess and check programs Π_{guess} and Π_{check}, respectively, into a single DLP Π_{solve} such that the answer sets of Π_{solve} yield the solutions of the overall problem.

(3) We demonstrate the method on the examples of QBFs and conformant planning [8] under fixed polynomial plan length (cf. [4,22]), where our method proves to loosen some restrictions of previous encodings.

Our work enlarges the range of techniques for expressing problems using ASP, in a direction which to our knowledge has not been explored so far. It also complements recent results about meta-interpretation in ASP [16,2,3]. We fruitfully exploit the construction of $tr(\cdot)$ to further elucidate the natural guess and check programming paradigm for ASP, as discussed in [11] or in [14] (named "Generate/Define/Test" there), and we fill a gap by providing an automated construction for integrating guess and check programs. It is worth noticing that such an integration is non-trivial even for manual construction in general. Apart from being pure ASP solutions, integrated encodings may be straight subject to automated program optimization within ASP solvers, considering both the guess and check part as well as their interaction; this is not immediate for separate programs.

For space constraints, most proofs and longer encodings are omitted here. All proofs and further details (encodings, etc) are given in an extended version of this paper.[1]

2 Preliminaries

We assume that the reader is familiar with logic programming and answer set semantics (see [7,19]) and only briefly recall the necessary concepts.

A *literal* is an atom $a(t_1, \ldots, t_n)$,or its negation $\neg a(t_1, \ldots, t_n)$, where "$\neg$" (alias, "–") is the strong negation symbol, in a function-free first-order language with at least one constant, which is customarily given by the programs considered. By $|a| = |\neg a| = a$ we denote the atom of a literal. Extended disjunctive logic programs (EDLPs; or simply programs) are finite sets Π of rules r

$$h_1 \texttt{ v } \ldots \texttt{ v } h_l \texttt{ :- } b_1, \ldots, b_m, \texttt{ not } b_{m+1}, \ldots \texttt{ not } b_n. \qquad (1)$$

$l, m, n \geq 0$, where each h_i and b_j is a literal and `not` is weak negation (negation as failure). By $H(r) = \{h_1, \ldots, h_l\}$, $B^+(r) = \{b_1, \ldots, b_m\}$, $B^-(r) = \{b_{m+1}, \ldots, b_n\}$, and $B(r) = B^+(r) \cup B^-(r)$ we denote the head and (pos., resp. neg.) body of rule r. Rules with $|H(r)|$=1 and $B(r)$=$\emptyset$ are *facts* and rules with $H(r)$=$\emptyset$ *constraints*. A rule r is *positive*, if "`not`" does not occur in it, and *normal*, if $|H(r)| \leq 1$. A program Π is *positive* (resp. *normal*) if all its rules are positive (resp., normal). We omit "extended" in what follows and refer to EDLPs as DLPs etc.

Literals (resp. rules, programs) are *ground* if they are variable-free. Non-ground rules (resp. programs) amount to their *ground instantiation*, i.e., all rules obtained by substituting variables with constants from the (implicit) language.

A ground program Π is *head-cycle free* [1], if no literals $l \neq l'$ occurring in the same rule head mutually depend on each other by positive recursion; Π is stratified [20,21], if no literal l depends by recursion through negation on itself.

Recall that the *answer set semantics* [7] for DLPs is as follows. Denote by $Lit(\Pi)$ the set of all ground literals for a program Π. Then, S is an *answer set* of Π, if S is a minimal (under $\subseteq$) consistent[2] set $S \subseteq Lit(\Pi)$ satisfying all rules in the reduct Π^S, which contains all rules $h_1 \texttt{ v } \ldots \texttt{ v } h_l \texttt{ :- } b_1, \ldots, b_m$ for all ground instances of rules (1) in Π such that $S \cap B^-(r) = \emptyset$.

[1] See http://www.kr.tuwien.ac.at/staff/axel/guessncheck/ (forthcoming).

[2] For our concerns, we disregard a possible inconsistent answer set.

3 Meta-interpreter Transformation

As mentioned above, a rewriting of a given program Π_{check} to a program Π'_{check} for integrating a guess and a check part into a single program is tricky in general. The working of the answer set semantics is not easy to be emulated in Π'_{check}, since essentially we lack negation in Π'_{check}: Upon a "guess" S for an answer set of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$, the reduct Π^S_{solve} is `not`-free. Thus, contrary to Π_{check}, there is no possibility to consider varying guesses for the value of negated atoms in Π'_{check} in combination with one guess for the negated atoms in Π_{guess} in the combined program Π_{solve}. On the other hand, if there is no disjunction in Π'_{check} then Π_{solve} is Horn; thus, its answer sets can be guessed and checked in NP.

This leads us to consider an approach in which the program Π'_{check} is constructed by the use of meta-interpretation techniques [16,2,3]: the idea is that a program Π is represented by a set of facts, $F(\Pi)$, which is input to a fixed program Π_{meta}, the meta-interpreter, such that the answer sets of $\Pi_{meta} \cup F(\Pi)$ correspond to the answer sets of Π. Note that existing meta-interpreters are normal logic programs, and can not be used for our purposes for the reasons explained above; we have to construct a novel meta-interpreter which is essentially `not`-free and contains disjunction. To this end, we exploit the following characterization of (consistent) answer sets for HDLPs:

Theorem 1 (cf. [1]). *For any ground HDLP Π, a consistent $S \subseteq Lit(\Pi)$ is an answer set of Π iff (1) S satisfies Π and (2) there is a function $\phi : Lit(\Pi) \to \{0, 1, 2, \dots\}$ such that for each literal $l \in S$, there is a rule $r \in \Pi$ with (a) $B^+(r) \subseteq S$, (b) $B^-(r) \cap S = \emptyset$, (c) $l \in H(r)$, (d) $S \cap (H(r) \setminus \{l\}) = \emptyset$, and (e) $\phi(l') < \phi(l)$ for each $l' \in B^+(r)$.*

Theorem 1 will now serve as a basis for a transformation from a given HDLP Π to a DLP $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$ such that $tr(\Pi)$ fulfills the properties **T1**–**T4**:

Input representation $F(\Pi)$. As input for the meta-interpreter Π_{meta} below, we choose the following representation $F(\Pi)$ of the propositional program Π.

We assume that each rule r has a unique name $n(r)$ as usual; for convenience, we identify r with $n(r)$. For any rule $r \in \Pi$, we set up in $F(\Pi)$ the following facts:

$\texttt{lit}(\texttt{h}, l, r).\ \texttt{atom}(l, \lvert l \rvert).$	for each literal $l \in H(r)$,
$\texttt{lit}(\texttt{p}, l, r).$	for each literal $l \in B^+(r)$,
$\texttt{lit}(\texttt{n}, l, r).$	for each literal $l \in B^-(r)$.

While the facts for predicate `lit` obviously encode the rules of Π, the facts for predicate `atom` indicate whether a literal is classically positive or negative. We only need this information for head literals; this will be further explained below.

Meta-interpreter Π_{meta}. We construct our meta-interpreter program Π_{meta}, which in essence is a positive disjunctive program, in a sequence of several steps. They center around checking whether a guess for an answer set $S \subseteq Lit(\Pi)$, encoded by a predicate $\texttt{inS}(\cdot)$, is an answer set of Π by testing the criteria of Theorem 1. The steps of the transformation cast the conditions of the theorem into rules of Π_{meta}, and provide auxiliary machinery for this aim.

Step 1 We add the following preprocessing rules:

```
 1: rule(L,R) :− lit(h,L,R), not lit(p,L,R), not lit(n,L,R).
 2: ruleBefore(L,R) :− rule(L,R), rule(L,R1), R1 < R.
 3: ruleAfter(L,R) :− rule(L,R), rule(L,R1), R < R1.
 4: ruleBetween(L,R1,R2) :− rule(L,R1), rule(L,R2), rule(L,R3),
                            R1 < R3, R3 < R2.
 5: firstRule(L,R) :− rule(L,R), not ruleBefore(L,R).
 6: lastRule(L,R) :− rule(L,R), not ruleAfter(L,R).
 7: nextRule(L,R1,R2) :− rule(L,R1), rule(L,R2), R1 < R2,
                         not ruleBetween(L,R1,R2).
 8: before(HPN,L,R) :− lit(HPN,L,R), lit(HPN,L1,R), L1 < L.
 9: after(HPN,L,R) :− lit(HPN,L,R), lit(HPN,L1,R), L < L1.
10: between(HPN,L,L2,R) :− lit(HPN,L,R), lit(HPN,L1,R),
                           lit(HPN,L2,R), L < L1, L1 < L2.
11: next(HPN,L,L1,R) :− lit(HPN,L,R), lit(HPN,L1,R), L < L1,
                        not between(HPN,L,L1,R).
12: first(HPN,L,R) :− lit(HPN,L,R), not before(HPN,L,R).
13: last(HPN,L,R) :− lit(HPN,L,R), not after(HPN,L,R).
14: hlit(L) :− rule(L,R).
```

Lines 1 to 7 fix an enumeration of the rules in Π from which a literal l may be derived, assuming a given order $<$ on rule names (e.g. in DLV [11], built-in lexicographic order; $<$ can also be easily generated using guessing rules). Note that under answer set semantics, we need only to consider rules where the literal l to prove does not occur in the body. Next, lines 8 to 13 fix enumerations of $H(r)$, $B^+(r)$ and $B^-(r)$ for each rule. The final line 14 collects all literals that can be derived from rule heads. Note that lines 1-14 plus $F(\Pi)$ form a stratified program, which has a single answer set (cf. [20,21]).

Step 2 We add rules which "guess" a candidate answer set $S \subseteq Lit(\Pi)$ and a total ordering `phi` on S corresponding with the function ϕ in Theorem 1.(2).

```
15: inS(L) v ninS(L):−hlit(L).
16: ninS(L):−lit(pn,L,R),not hlit(L).          for each pn ∈ {p,n}
17: notok:−inS(L), inS(NL), L != NL, atom(L,A), atom(NL,A).
18: phi(L,L1) v phi(L1,L):−inS(L), inS(L1), L < L1.
19: phi(L,L2):−phi(L,L1), phi(L1,L2).
```

Line 15 focuses the guess of S to literals occurring in some relevant rule head in Π; other literals can not belong to S (line 16). Line 17 then checks whether S is consistent, deriving a new distinct atom `notok` otherwise. Line 18 guesses a strict total order `phi` on `inS` where line 19 guarantees transitivity; note that minimality of answer sets prevents that `phi` is cyclic, i.e., that `phi(L,L)` holds.

In the subsequent steps, we check whether S and `phi` violate the conditions of Theorem 1 by deriving the distinct atom `notok` in case, indicating that S is not an answer set or `phi` does not represent a proper function ϕ.

Step 3 Corresponding to condition 1 in Theorem 1, notok is derived whenever there is an unsatisfied rule by the following program part:

```
20: allInSUpto(p,Min,R):−inS(Min),first(p,Min,R).
21: allInSUpto(p,L1,R):− inS(L1),allInSUpto(p,L,R),next(p,L,L1,R).
22: allInS(p,R):−allInSUpto(p,Max,R),last(p,Max,R).
23: allNinSUpto(hn,Min,R):−ninS(Min),first(hn,Min,R).
24: allNinSUpto(hn,L1,R):− ninS(L1),allNinSUpto(hn,L,R),
                            next(hn,L,L1,R).
25: allNinS(hn,R):−allNinSUpto(hn,Max,R),last(hn,Max,R).
                                                  } for each hn ∈ {h,n}
26: hasHead(R):−lit(h,L,R).
27: hasPBody(R):−lit(p,L,R).
28: hasNBody(R):−lit(n,L,R).
29: allNinS(h,R):−lit(HPN,L,R),not hasHead(R).
30: allInS(p,R):−lit(HPN,L,R),not hasPBody(R).
31: allNinS(n,R):−lit(HPN,L,R),not hasNBody(R).
32: notok:−allNinS(h,R),allInS(p,R),allNinS(n,R),lit(HPN,L,R).
```

These rules compute by iteration over $B^+(r)$ (resp. $H(r)$, $B^-(r)$) for each rule r, whether for all positive body (resp. head and weakly negated body) literals in rule r inS holds (resp. ninS holds) (lines 20 to 25). Here, empty heads (resp. bodies) are interpreted as unsatisfied (resp. satisfied), cf. lines 26 to 31. The final rule 32 fires exactly if one of the original rules from Π is unsatisfied.

Step 4 We derive notok whenever there is a literal $l \in S$ which is not provable by any rule r wrt. phi. This corresponds to checking condition 2 from Theorem 1.

```
33: failsToProve(L,R):−rule(L,R),lit(p,L1,R),ninS(L1).
34: failsToProve(L,R):−rule(L,R),lit(n,L1,R),inS(L1).
35: failsToProve(L,R):−rule(L,R),rule(L1,R),inS(L1),L1 != L,inS(L).
36: failsToProve(L,R):−rule(L,R),lit(p,L1,R),phi(L1,L).
37: allFailUpto(L,R):−failsToProve(L,R),firstRule(L,R).
38: allFailUpto(L,R1):− failsToProve(L,R1),allFailUpto(L,R),
                        nextRule(L,R,R1).
39: notok:−allFailUpto(L,R),lastRule(L,R),inS(L).
```

Lines 33 and 34 check whether condition $2.(a)$ or (b) are violated, i.e. some rule can only prove a literal if its body is satisfied. Condition $2.(d)$ is checked in line 35, i.e. r fails to prove l if there is some $l' \neq l$ such that $l' \in H(r) \cap S$. Violations of condition $2.(e)$ are checked in line 36. Finally, lines 37 to 39 derive notok if all rules fail to prove some literal $l \in S$ by iterating over all rules with $l \in H(r)$ using the order from Step 1. Thus, condition $2.(c)$ is implicitly checked.

Step 5 Whenever notok is derived, indicating a wrong guess, then we apply a saturation technique as in [6,12] to some other predicates, such that a canonical set Ω results. This set turns out to be an answer set iff no guess for S and ϕ works out, i.e., Π has no answer set. In particular, we saturate the predicates inS, ninS, and phi by the following rules:

40: $\texttt{phi(L,L1)}\text{:}-\texttt{notok}, \texttt{hlit(L)}, \texttt{hlit(L1)}.$
41: $\texttt{inS(L)}\text{:}-\texttt{notok}, \texttt{hlit(L)}.$
42: $\texttt{ninS(L)}\text{:}-\texttt{notok}, \texttt{hlit(L)}.$

Intuitively, by these rules, any answer set containing `notok` is "blown up" to an answer set Ω containing all possible guesses for `inS`, `ninS`, and `phi`.

3.1 Answer Set Correspondence

Let $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$, where $F(\Pi)$ and Π_{meta} are the input representation and meta-interpreter as defined above. Clearly, $tr(\Pi)$ satisfies property **T3**, and as easily checked, $tr(\Pi)$ is modular. Moreover, $\neg$ does not occur in $tr(\Pi)$ and `not` only stratified. The latter is not applied to literals depending on disjunction; it thus occurs only in the deterministic part of $tr(\Pi)$, i.e. **T4** holds.

To establish **T1** and **T2**, we define the literal set Ω as follows:

Definition 1. *Let Π^i_{meta} be the set of rules in Π_{meta} established in Step $i \in \{1, \ldots, 5\}$. For any program Π, let $\Pi_\Omega = F(\Pi) \cup \bigcup_{i \in \{1,3,4,5\}} \Pi^i_{meta} \cup \{\texttt{notok}.\}$. Then, Ω is defined as the answer set of Π_Ω.*

The fact that Π_Ω is a stratified normal logic program without $\neg$ and constraints, which as well-known has a single answer set, yields the following lemma.

Lemma 1. *Ω is well-defined and uniquely determined by Π.*

Theorem 2. *For any given HDLP Π the following holds for $tr(\Pi)$:*

1. *$tr(\Pi)$ has some answer set, and $S' \subseteq \Omega$ for any answer set S' of $tr(\Pi)$.*
2. *S is an answer set of Π if and only if there exists an answer set S' of $tr(\Pi)$ such that $S = \{l \mid \texttt{inS}(l) \in S'\}$ and $\texttt{notok} \notin S'$.*
3. *Π has no answer set if and only if $tr(\Pi)$ has the unique answer set Ω.*

The following proposition is not difficult to establish.

Proposition 1. *Given Π, the transformation $tr(\Pi)$, as well as the ground instantiation of $tr(\Pi)$, is computable in* LOGSPACE *(thus in polynomial time).*

Note that $tr(\Pi)$ is not polynomial faithful modular (PFM) in the sense of [9]: (i) **T1** does not claim a strict one-to-one correspondence between the answer sets of Π and $tr(\Pi)$. Indeed, $tr(\Pi)$ might have several answer sets corresponding to a particular answer set S of Π, reflecting different possible guesses for ϕ. (ii) Faithfulness as in [9] conflicts with property **T2**.

As noticed above, $tr(\Pi)$ uses weak negation only stratified and in a deterministic part of the program; we can easily eliminate it by computing in the transformation the complement of each predicate accessed through `not` and providing it in $F(\Pi)$ as facts; we then obtain a positive program. (The built-in predicates $<$ and != can be eliminated similarly if desired.) However, this modified transformation is not modular. As shown next, this is not incidental.

Proposition 2. *There is no modular transformation $tr'(\Pi)$ from HDLPs to DLPs satisfying* **T1**, **T2** *and* **T3** *such that $tr'(\Pi)$ is a positive program.*

Proof. Assuming that such a transformation $tr'(\Pi)$ exists, we derive a contradiction. Let $\Pi_1 = \{\mathtt{a} :\!- \ \mathtt{not\ b}.\}$ and $\Pi_2 = \Pi_1 \cup \{\mathtt{b}.\}$. Then, $tr'(\Pi_2)$ has some answer set S_2. Since $tr'(\cdot)$ is modular, $tr'(\Pi_1) \subseteq tr'(\Pi_2)$ holds and thus S_2 satisfies each rule in $tr'(\Pi_1)$. Hence, S_2 contains some answer set S_1. By **T1**, $\mathtt{inS}(\mathtt{a}) \in S_1$ must hold, and hence $\mathtt{inS}(\mathtt{a}) \in S_2$. By **T1** again, it follows that Π_2 has an answer set S such that $\mathtt{a} \in S$. But the single answer set of Π_2 is $\{\mathtt{b}\}$, a contradiction. □

We remark that Proposition 2 remains true if **T1** is generalized such that the answer set S of Π corresponding to S' is given by $S = \{l \mid S' \models \Phi(l)\}$, where $\Phi(x)$ is a monotone query (e.g., computed by a normal positive program without constraints). Moreover, if a successor predicate $\mathtt{next}(\mathtt{X}, \mathtt{Y})$ and predicates $\mathtt{first}(\mathtt{X})$ and $\mathtt{last}(\mathtt{X})$ for the constants are available (on a finite universe, resp. the constants in Π and rule names), then the negation of the non-input predicates accessed through `not` can be computed by a positive normal program, since such programs capture polynomial time computability by well-known results on the expressive power of Datalog [18]; thus, negation of input predicates in $F(\Pi)$ is sufficient in this case.

3.2 Optimizations

Π_{meta} can be modified in several respects. We discuss here some modifications which, though not necessarily shrinking size of the transformation, intuitively prune the search of an answer set solver applied to $tr(\Pi)$. The extended paper considers further ones.

(OPT1) *Give up modularity* If we sacrifice modularity (i.e. that $tr(\Pi) = \bigcup_{r \in \Pi} tr(r)$), and allow that Π_{meta} partly depends on the input, then we can circumvent the iterations in Step 3 and part of Step 1 as follows: We substitute Step 3 by rules

$$\mathtt{notok} :\!- \ \mathtt{ninS}(h_1), \ldots, \mathtt{ninS}(h_l), \ \mathtt{inS}(b_1), \ldots, \mathtt{inS}(b_m), \mathtt{ninS}(b_{m+1}), \ldots \mathtt{ninS}(b_n). \tag{2}$$

for each rule r in Π of form (1). These rules can be efficiently generated in parallel to $F(\Pi)$. Lines 8 to 13 of Step 1 then can also be dropped.

We can even refine this further. For any normal rule $r \in \Pi$ with $|H(r)| = 1$ which has a satisfied body, we can force the guess of h: we replace (2) by

$$\mathtt{inS}(h) :\!- \ \mathtt{inS}(b_1), \ \ldots, \ \mathtt{inS}(b_m), \mathtt{ninS}(b_{m+1}), \ \ldots \ \mathtt{ninS}(b_n). \tag{3}$$

In this context, since constraints only serve to "discard" unwanted models but cannot prove any literal, we can ignore them during input generation $F(\Pi)$; rule (2) is sufficient. Note that dropping input representation $\mathtt{lit}(\mathtt{n}, l, c).$ for literals only occurring in the negative body of constraints but nowhere else in Π requires some care. Such l can be removed by simple preprocessing, though.

(OPT2) *Optimize guess of order.* We only need to guess and check the order ϕ for literals L, L' if they allow for cyclic dependency, i.e., they appear in the heads of rules within the same strongly connected component of the program wrt. S.[3] These dependencies wrt. S are easily computed:

$\texttt{dep(L,L1)}\text{:}-\texttt{lit(h,L,R)}, \texttt{lit(p,L1,R)}, \texttt{inS(L)}, \texttt{inS(L1)}.$
$\texttt{dep(L,L2)}\text{:}-\texttt{lit(h,L,R)}, \texttt{lit(p,L1,R)}, \texttt{dep(L1,L2)}, \texttt{inS(L)}.$
$\texttt{cyclic}\text{:}-\texttt{dep(L,L1)}, \texttt{dep(L1,L)}.$

The guessing rules for ϕ (line 18 and 19) are then be replaced by:

$\texttt{phi(L,L1)} \texttt{ v } \texttt{phi(L,L1)}\text{:}-\texttt{dep(L,L1)}, \texttt{dep(L1,L)}, \texttt{L} < \texttt{L1}, \texttt{cyclic}.$
$\texttt{phi(L,L2)}\text{:}-\texttt{phi(L,L1)}, \texttt{phi(L1,L2)}, \texttt{cyclic}.$

Moreover, we add the new atom `cyclic` also to the body of the rules where `phi` appears (lines 36,40) to check `phi` only if Π has *any* cyclic dependencies wrt. S.

4 Integrating Guess and co-NP Check Programs

A general method for solving NP problems using answer set programming is given by the so called "guess and check" paradigm: First a (possibly disjunctive) program is used to guess a set of candidate solutions, and then rules and constraints are added which eliminate unwanted solutions. DLPs allow for the formulation of such problems in a very intuitive way (e.g. solutions of 3-colorability, deterministic planning, etc.) if checking is easy (often polynomial), such as checking whether no adjacent nodes have the same color, a course of deterministic actions reaches a certain goal, etc. For instance, given a graph as a set of facts of the form $\texttt{node}(x)$. and $\texttt{edge}(x, y)$. we can write a simple DLP which guesses and checks all possible 3-colorings as follows:

$\texttt{col(red,X) v col(green,X) v col(blue,X)} \text{:}-\ \texttt{node(X)}.$ } *Guess*
$\text{:}-\texttt{edge(X,Y)}, \texttt{col(C,X)}, \texttt{col(C,Y)}.$ } *Check*

However, encoding problems where the check is in co-NP but not known to be polynomial (or in NP) is not always obvious (e.g., for conformant planning [4], or minimal update answer sets [5]). A simple, common workaround is to write two programs:

(i) a normal LP or HDLP Π_{guess}, which guesses some solution;
(ii) a HDLP (equivalently, normal LP) Π_{check} which encodes the co-NP check,

and proceed as follows: First compute, one by one, the candidate solutions $S_1, S_2, \ldots$ as answer sets of Π_{guess}; then, pipe each S_i as input to Π_{check}; finally, output S_i if $\Pi_{check} \cup S_i$ has no answer set.

By the computational power of full disjunctive logic programs (Σ_2^P [6]), we know that such problems can also be expressed by a single EDLP, Π_{solve}. In the following, we show how our transformation tr resp. tr_{Opt} from above can be used to automatically combine Π_{guess} and Π_{check} into a single program.

[3] Similarly, in [1] $\phi : Lit(\Pi) \rightarrow \{1, \ldots, r\}$ is only defined for a range r bound by the longest acyclic path in any strongly connected component of the program.

We assume that the set $Lit(\Pi_{guess})$ is a Splitting Set [13] of $\Pi_{guess} \cup \Pi_{check}$, i.e. no head literal from Π_{check} occurs in Π_{guess}. This can be easily achieved by introducing new predicate names, e.g., $\mathtt{p}'$ for a predicate $\mathtt{p}$, and adding a rule $\mathtt{p}'(t){:-}\mathtt{p}(t)$ in case. Each rule r in Π_{check} is of the form

$$\begin{aligned} h_1 \ \mathtt{v} \cdots \ \mathtt{v}\, h_l \ \ {:-} \ \ & bc_1,\ \ldots,\ bc_m,\ \mathtt{not}\ bc_{m+1},\ \ldots,\ \mathtt{not}\ bc_n \\ & bg_1,\ \ldots,\ bg_p,\ \mathtt{not}\ bg_{p+1},\ \ldots,\ \mathtt{not}\ bg_q. \end{aligned} \tag{4}$$

where the bg_i are the body literals defined in Π_{guess}. We write $\mathsf{B}_{guess}(r)$ for $bg_1, \ldots, bg_p$, $\mathtt{not}\ bg_{p+1}, \ldots, \mathtt{not}\ bg_q$. We now define a new check program.

Program Π'_{check} contains the following rules and constraints:

1. The facts $F(\Pi_{check})$ in a conditional version: For each $r \in \Pi_{check}$ of form (4),

$$\begin{array}{rll} \mathtt{lit}(\mathtt{h}, l, r) \ {:-}\ \mathsf{B}_{\mathtt{guess}}(r). & \mathtt{atom}(l, |l|). & \text{for each } l \in H(r); \\ \mathtt{lit}(\mathtt{p}, bc_i, r) \ {:-}\ \mathsf{B}_{\mathtt{guess}}(r). & & \text{for each } i \in \{1, \ldots, m\}; \\ \mathtt{lit}(\mathtt{n}, bc_j, r) \ {:-}\ \mathsf{B}_{\mathtt{guess}}(r). & & \text{for each } j \in \{m+1, \ldots, n\}. \end{array}$$

2. each rule in Π_{meta} (where for the optimized version, in (2) and (3) $\mathsf{B}_{guess}(r)$ is added to the bodies);
3. finally, a constraint ${:-}\ \mathtt{not}\ \mathtt{notok}$. This will eliminate all answer sets S such that $\Pi_{check} \cup S$ has an answer set.

The union of Π_{guess} and Π'_{check} then amounts to the desired integrated encoding Π_{solve}, which is expressed by the following result.

Theorem 3. *For Π_{guess} and Π_{check}, the answer sets S' of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$ correspond 1-1 with the answer sets S of Π_{guess} s.t. $\Pi_{check} \cup S$ has no answer set.*

Note that integrating guess and check programs Π_{guess} and Π_{check}, respectively, to succeed iff $\Pi_{check} \cup S$ has *some* answer set, is easy. After ensuring the Splitting Set property (if needed), simply take $\Pi_{solve} = \Pi_{guess} \cup \Pi_{check}$; its answer sets correspond on the predicates in Π_{guess} to the desired solutions.

5 Applications

We now exemplify the use of our transformation for two Σ_2^P-complete problems, which thus involve co-NP-complete solution checking: one is about Quantified Boolean formulas (QBFs) with one quantifier alternation, which are well-studied in Answer Set Programming, and the other about conformant planning [4,22]. Further examples of such problems can be found e.g. in [6,5,11] (and solved similarly). However, note that our method is applicable to *any* checks encoded by inconsistency of some HDLP program; co-NP-hardness is not a prerequisite.

5.1 Quantified Boolean Formulas

Given a QBF $F = \exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n \, \Phi$, where $\Phi = c_1 \vee \cdots \vee c_k$ is a propositional formula over $x_1, \ldots, x_m, y_1, \ldots, y_n$ in disjunctive normal form, i.e. each $c_i = \ell_{i,1} \wedge \cdots \wedge \ell_{i,i_l}$ and $|\ell_{i,j}| \in \{x_1, \ldots, x_m, y_1 \ldots, y_n\}$, compute the assignments to the variable $x_1, \ldots, x_m$ which witness that F evaluates to true.

Intuitively, this problem can be solved by "guessing and checking" as follows:

(QBF_g**)** Guess a truth assignment for the variables $x_1, \ldots, x_m$.
(QBF_c**)** Check whether this assignment satisfies Φ for all assignments of $y_1, \ldots, y_n$.

Both parts can be encoded by very simple HDLPs:

QBF_g :

x_1 v $-x_1$. ... x_m v $-x_m$.

QBF_c :

y_1 v $-y_1$. ... y_n v $-y_n$.
:− $\ell_{1,1}, \ldots, \ell_{1,1_l}$. ... :− $\ell_{k,1}, \ldots, \ell_{k,k_l}$.

Obviously, for any answer set S of QBF_g, representing an assignment to $x_1, \ldots, x_n$, the program $QBF_c \cup S$ has no answer set thanks to the constraints, iff every assignment for $y_1, \ldots, y_n$ satisfies formula Φ then. By the method sketched, we can now automatically generate a single program QBF_{solve} integrating the guess and check programs (cf. Footnote 1). Note that the customary (but tricky) saturation technique to solve this problem (cf. [6,11]) is fully transparent to the non-expert.

5.2 Conformant Planning

Loosely speaking, planning is the problem to find a sequence of actions $P = \alpha_1, \alpha_2, \ldots, \alpha_n$, a *plan*, which takes a system from an initial state s_0 to a state s_n in which a goal (often, given by an atom g) holds, where a state s is described by values of fluents, i.e., predicates which might change over time. *Conformant planning* [8] is concerned with finding a plan P which works under all contingencies that may arise from incomplete information about the initial state and/or nondeterministic action effects, which is in Σ_2^P under certain restrictions, cf. [4,22]. Hence, the problem can be solved with a guess and (co-NP) check strategy.

As an example, we consider a simplified version of the well-known "*Bomb in the Toilet*" planning problem (cf. [4,17]): We have been alarmed that a possibly armed bomb is in a lavatory which has a toilet bowl. Possible actions are dunking the bomb into the bowl and flushing the toilet. After just dunking, the bomb may be disarmed or not; only flushing the toilet guarantees that it is really disarmed.

Using the following guess and check programs $Bomb_g$ and $Bomb_c$, respectively, we can compute a plan for having the bomb disarmed by two actions:

$Bomb_g$:

% Timestamps:
`time(0). time(1).`
%Guess a plan:
`dunk(T) v -dunk(T):−time(T).`
`flush(T) v -flush(T):−time(T).`

$Bomb_c$:

% Initial state:
`armed(0) v -armed(0).`
% Frame Axioms:
`armed(T1):−armed(T), time(T),`
`not -armed(T1), T1 = T + 1.`

```
% Forbid concurrent actions:
:− flush(T), dunk(T).

dunked(T1):−dunked(T), T1 = T + 1.
% Effect of dunking:
dunked(T1):−dunk(T), T1 = T + 1.
armed(T1) v -armed(T1):−dunk(T),
                        armed(T), T1 = T + 1.
% Effect of flushing:
 -armed(T1):−flush(T), dunked(T), T1 = T + 1.
% Check goal in stage 2:
:− not armed(2).
```

$Bomb_g$ guesses all candidate plans $P = \alpha_1, \alpha_2$, using time points for action execution, while $Bomb_c$ checks whether any such plan P is conformant for the goal g = `not armed(2)`. Here, absence of `armed`(t) is viewed as `-armed`(t), i.e. CWA is used, which saves a negative frame axiom on `-armed`. The final constraint eliminates a plan execution iff it reaches the goal; thus, $Bomb_c$ has no answer set iff the plan P is conformant. Answer set $S = \{$`time(0)`, `time(1)`, `dunk(0)`, `flush(1)`$\}$ of $Bomb_g$ corresponds to the (single) conformant plan P= `dunk`, `flush` for goal `not armed(2)`.

By our general method, $Bomb_g$ and $Bomb_c$ can be integrated automatically into a single program $Bomb_{plan} = Bomb_g \cup Bomb_c{}'$ (cf. Footnote 1). It has a single answer set, corresponding to the single conformant plan P= `dunk`, `flush` as desired.

Note that our rewriting method is more generally applicable than the encoding for conformant planning proposed by Leone *et al.* [12] who require that state transitions are specified by a positive constraint-free LP. Our method can still safely be used in presence of negation and constraints, provided action execution always leads to a consistent successor state (cf. [4,22] for a discussion).

6 Experiments

We have conducted some experiments to get an idea about the performance of our automatically integrated encodings (even though this was not the major concern). To this end, we made comparisons to hand-written integrated encodings (in particular, for QBF evaluation using an ad hoc encoding from [11]) and to interleaved guess and check programs (in particular, for conformant planning on variants of the Bomb in the Toilet problem, executed on the DLV$^{\mathcal{K}}$ planning system [4]). The results are shown in Table 1.

We have considered some random QBF instances with n existential and n universal variables, denoted as QBF-n. For conformant planning, we have compared the integrated encodings for some "Bomb in the Toilet" problems described in [4], where the names BTC(i), BTUC(i) are defined, against the DLV$^{\mathcal{K}}$ planning system [4], which implements conformant planning by interleaved calls to separate guess and check programs. The columns in Table 1 report times for encodings using the basic approach ($meta$), optimizations $OPT1$, $OPT2$ and both (OPT), respectively, compared with the ad hoc encoding for QBFs and interleaved conformant plan computation using DLV$^{\mathcal{K}}$; a dash marks exceeding a time limit of 300s (QBFs), resp. 4000s (conformant planning). We used DLV [4] as a platform since other disjunctive ASP engines, in particular GNT,[5] were

[4] http://www.dlvsystem.com

[5] http://www.tcs.hut.fi/Software/gnt/

Table 1. Experimental results for QBF (left) and Conformant Planning (right) for Bomb in Toilet

	$adhoc$ [11]	$meta$	$OPT1$	$OPT2$	OPT
QBF-4	0.01s	0.15s	0.10s	0.08s	0.06s
QBF-6	0.01s	1.08s	0.36s	0.17s	0.08s
QBF-8	0.01s	10.42s	1.43s	0.46s	0.10s
QBF-10	0.01s	60.74s	2.65s	1.32s	0.10s
QBF-12	0.01s	-	-	5.59s	0.11s
QBF-16	0.08s	-	-	-	0.50s

Average times for 10 randomly chosen instances per size.

	DLV$^{\mathcal{K}}$ [4]	$meta$	$OPT1$	$OPT2$	OPT
BTC(2)	0.01s	1.16s	0.80s	0.15s	0.08s
BTC(3)	0.11s	9.33s	9.25s	8.18s	4.95s
BTC(4)	4.68s	71.3s	67.8s	333s	256s
BTUC(2)	0.01s	6.38s	6.26s	0.22s	0.17s
BTUC(3)	1.78s	-	-	28.12s	13.0s
BTUC(4)	577s	-	-	-	2322s

significantly slower on all tested instances. More details and experiments are given in the extended paper (cf. Footnote 1).

Clearly, the performance of the automatic integrated encodings was expected to stay behind the other methods. Interestingly, for the QBF problem, the performance of our optimized translation stays within reach of the ad hoc encoding in [11] for small instances. For the planning problems, the integrated encodings tested still stay behind the interleaved computation of DLV$^{\mathcal{K}}$.

The results obtained using DLV show that the "guess and saturate" strategy in our approach benefits a lot from optimizations, but it might depend on the structure of Π_{guess} and Π_{check}, as well as on the heuristics used by DLV, which modifications yield gains. We strongly believe that there is room for further improvements both on the translation and for the underlying DLV engine. We emphasize that the strength of our approach appears if an integrated ad hoc encoding is non-obvious. Then, by our method such an encoding can be generated automatically from separate guess and check programs, which are often easy to formalize, while a manual integrated encoding may be difficult to find (as in the case of conformant planning or minimal update answer sets [5]).

7 Conclusion

We presented a method for rewriting a head-cycle free (extended) disjunctive logic program (HDLP) Π into a stratified disjunctive logic program without constraints $tr(\Pi)$, such that their answer sets correspond and a designated answer set of $tr(\Pi)$ indicates inconsistency of Π. Moreover, we showed how to use this method for automatically integrating a guess and separate check program for a co-NP property (expressed by inconsistency of an HDLP), into an equivalent single (extended) disjunctive logic program. This reconciles pragmatic problem solving with the natural "guess and check" resp. "Generate/Define/Test" approach in Answer Set Programming [11,14], in case a single program expressing the problem is difficult to write. In particular, it relieves the ASP user from using tricky saturation techniques, as customary e.g. for QBF evaluation.

We consider our work as an initial step for further research on automatic integration of guess and check encodings which exploits the full expressive power of DLPs: Integrated encodings like those considered are infeasible in less expressive frameworks such as propositional SAT solving or normal logic programming. However, while ad hoc encodings for specific problems are available, general methods in this direction were still missing.

Several issues remain for further work. Our rewriting method currently applies to propositional programs. Thus, before transformation, the program should be instantiated. A more efficient extension of the method to non-ground programs is needed, as well as

further improvements to the current transformations. Experimental results suggest that structural analysis of the guess and check programs might be valuable for this.

A further issue are alternative transformations, possibly tailored for certain classes of programs. Ben-Eliyahu and Dechter's work [1], on which we build, aimed at transforming HDLPs to SAT problems. It might be interesting to investigate whether related methods such as the one developed for ASSAT [15], which was recently generalized by Lee and Lifschitz [10] to disjunctive programs, can be adapted for our approach.

References

1. R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
2. J. Delgrande, T. Schaub, H. Tompits. plp: A generic compiler for ordered logic programs. *Proc. LPNMR'01*, LNCS 2173, pp. 411–415. Springer, 2001.
3. T. Eiter, W. Faber, N. Leone, G. Pfeifer. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory & Practice of Logic Progr.*, 3(4-5):463–498.
4. T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres. A logic programming approach to knowledge-state planning, II: The DLV^{K} system. *Artif. Intell.*, 144(1-2):157-211, 2003.
5. T. Eiter, M. Fink, G. Sabbatini, H. Tompits. On properties of update sequences based on causal rejection. *Theory & Practice of Logic Prog.*, 2(6):721–777, 2002.
6. T. Eiter, G. Gottlob, H. Mannila. Disjunctive datalog. *ACM TODS*, 22(3):364–418, 1997.
7. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
8. R. Goldman and M. Boddy. Expressive planning and explicit knowledge. *Proc. 3rd Int'l Conf. on AI Planning and Scheduling (AIPS-96)*, pp. 110–117, 1996.
9. T. Janhunen. On the effect of default negation on the expressiveness of disjunctive rules. *LPNMR'01*, LNCS 2173, pp. 93–106. Springer, 2001.
10. J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. 19th Int'l Conf. on Logic Programming (ICLP-03)*, December 2003. To appear.
11. N. Leone, G. Pfeifer, W. Faber *et al.* The DLV system for knowledge representation and reasoning. Tech. Rep. INFSYS RR-1843-02-14, Information Sys. Institute, TU Wien, 2002.
12. N. Leone, R. Rosati, F. Scarcello. Enhancing answer set planning. *Proc. IJCAI-01 Workshop on Planning under Uncertainty & Incomplete Information*, pp. 33–42, 2001.
13. V. Lifschitz and H. Turner. Splitting a logic program. *Proc. ICLP'94*, pp. 23–37, 1994.
14. V. Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138:39–54, 2002.
15. F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Proc. 18th National Conf. on Artificial Intelligence (AAAI-2002)*, 2002.
16. V.W. Marek and J.B. Remmel. On the Expressibility of stable logic programming. *Proc. LPNMR'01*, LNCS 2173, pp. 107–120. Springer, 2001.
17. D. McDermott. A critique of pure reason. *Computational Intelligence*, 3:151–237, 1987.
18. C. H. Papadimitriou. A note on the expressive power of Prolog. *Bulletin of the EATCS*, 26:21–23, 1985.
19. A. Provetti and T.C. Son, editors. *Proc. AAAI 2001 Spring Symposium on Answer Set Programming*, Stanford, CA, March 2001. Workshop Technical Report SS-01-01, AAAI Press.
20. T. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.
21. T. Przymusinski. Stable semantics for disjunctive programs. *New Gen. Comp.*, 9, 1991.
22. H. Turner. Polynomial-length planning spans the Polynomial Hierarchy. *Proc. 8th European Conf. on Artificial Intelligence (JELIA 2002)*, LNCS 2424, pp. 111–124. Springer, 2002.

Graphs and Colorings for Answer Set Programming: Abridged Report

Kathrin Konczak, Thomas Linke, and Torsten Schaub*

Institut für Informatik, Universität Potsdam, Postfach 90 03 27, D–14439 Potsdam

Abstract. We investigate rule dependency graphs and their colorings for characterizing the computation of answer sets of logic programs. We start from a characterization of answer sets in terms of totally colored dependency graphs. To a turn, we develop a series of operational characterizations of answer sets in terms of operators on partial colorings. In analogy to the notion of a derivation in proof theory, our operational characterizations are expressed as (non-deterministically formed) sequences of colorings, turning an uncolored graph into a totally colored one. This results in an operational framework in which different combinations of operators result in different formal properties. Among others, we identify the basic strategy employed by the `noMoRe` system and justify its algorithmic approach. Also, we distinguish Fitting's and well-founded semantics.

1 Introduction

We elaborate upon using graphs for characterizing the computation of answer sets of logic programs. To this end, we build upon the theoretical foundations introduced in [9, 1]. Accordingly, we are interested in characterizing answer sets by means of their set of generating rules. For determining, whether a rule belongs to this set, we must verify that each positive body atom is derivable and that no negative body atom is derivable. In fact, an atom is derivable, if the set of generating rules includes a rule, having the atom as its head; or conversely, an atom is not derivable, if there is no rule among the generating rules that has the atom as its head. Consequently, the formation of the set of generating rules boils down to resolving positive and negative dependencies among rules. For capturing these dependencies, we take advantage of the concept of a *rule dependency graph*, wherein each node represents a rule in the underlying program and two types of edges stand for the aforementioned positive and negative rule dependencies, respectively. For expressing the applicability status of rules, that is, whether a rule belongs to a set of generating rules or not, we *color* the respective nodes in the graph. In this way, an answer set can be expressed by a total coloring of the rule dependency graph. Of course, in what follows, we are mainly interested in the inverse, that is, when does a graph coloring correspond to an answer set; and, in particular, how can we compute such a total coloring. To this end, we start by identifying graph structures that allow for characterizing answer sets in terms of totally colored dependency graphs. We then build upon these characterizations for developing an operational framework for answer set formation. The idea is to start from an uncolored rule dependency graph and to employ

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 127–140, 2004.

specific operators that turn a partially colored graph gradually into a totally colored one that represents an answer set. This approach is strongly inspired by the concept of a derivation, in particular, that of an SLD-derivation [14]. Accordingly, a program has a certain answer set iff there is a sequence of operations turning the uncolored graph into a totally colored one, expressing the answer set.

2 Rules, Programs, Graphs, and Colorings

A *logic program* is a finite set of rules such as $p_0 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$, where $n \geq m \geq 0$, and each p_i $(0 \leq i \leq n)$ is an *atom*. For such a rule r, we let $head(r)$ denote the *head*, p_0, of r and $body(r)$ the *body*, $\{p_1, \ldots, p_m,\ not\ p_{m+1}, \ldots, not\ p_n\}$, of r. Let $body^+(r) = \{p_1, \ldots, p_m\}$ and $body^-(r) = \{p_{m+1}, \ldots, p_n\}$. A program is *basic* if $body^-(r) = \emptyset$ for all its rules. The *reduct*, Π^X, of a program Π *relative to* a set X of atoms is defined by $\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}$. A set of atoms X is *closed under* a basic program Π if for any $r \in \Pi$, $head(r) \in X$ if $body^+(r) \subseteq X$. The smallest set of atoms being closed under a basic program Π is denoted by $Cn(\Pi)$. Then, a set X of atoms is an *answer set* of a program Π if $Cn(\Pi^X) = X$. We use $AS(\Pi)$ for denoting the set of all answer sets of Π. The set of *generating rules* of a set X of atoms from program Π is given by $R_\Pi(X) = \{r \in \Pi \mid body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}$.

Next, we lay the graph-theoretical foundations of our approach. A *graph* is a pair (V, E) where V is a set of *vertices* and $E \subseteq V \times V$ a set of (directed) *edges*. Labeled graphs posses multiple sets of edges. A graph (V, E) is *acyclic* if E contains no cycles. For $W \subseteq V$, we denote $E \cap (W \times W)$ by $E|_W$ and abbreviate $G = (V \cap W, E|_W)$ by $G|_W$. A *subgraph* of (V, E) is a graph (W, F) such that $W \subseteq V$ and $F \subseteq E|_W$.

In the sequel, we are interested in graphs reflecting dependencies among rules.

Definition 1. *Let Π be a logic program. The rule dependency graph (RDG) $\Gamma_\Pi = (\Pi, E_0, E_1)$ of Π is a labeled directed graph with*

$$E_0 = \left\{(r, r') \mid r, r' \in \Pi, head(r) \in body^+(r')\right\};$$
$$E_1 = \left\{(r, r') \mid r, r' \in \Pi, head(r) \in body^-(r')\right\}.$$

We omit the subscript Π from Γ_Π whenever the underlying program is clear from the context. An i-subgraph (V, E) of Γ is a subgraph of Γ with $E \subseteq E_i$ for $i \in \{0, 1\}$.

For illustration, consider the logic program $\Pi_1 = \{r_1, \ldots, r_6\}$, where

$$\begin{array}{lll} r_1 : p \leftarrow & r_3 : f \leftarrow b, not\ f' & r_5 : b \leftarrow m \\ r_2 : b \leftarrow p & r_4 : f' \leftarrow p, not\ f & r_6 : x \leftarrow f, f', not\ x \end{array} \quad (1)$$

The *RDG* of Π_1 is depicted graphically in Figure 1a. The *RDG* Γ_{Π_1} has among others 0-subgraph $(\{r_1, \ldots, r_4\}, \{(r_1, r_2)\})$ and 1-subgraph $(\{r_5, r_6\}, \{(r_6, r_6)\})$.

We call C a *coloring* of Γ_Π if C is a mapping $C : \Pi \to \{\oplus, \ominus\}$. We denote the set of all partial colorings of a *RDG* Γ_Π by $\mathbb{C}_{\Gamma_\Pi}$. For readability, we often omit the index Γ_Π. Intuitively, the colors $\oplus$ and $\ominus$ indicate whether a rule is supposedly applied or blocked. We define $C_\oplus = \{r \mid C(r) = \oplus\}$ and $C_\ominus = \{r \mid C(r) = \ominus\}$ for obtaining all

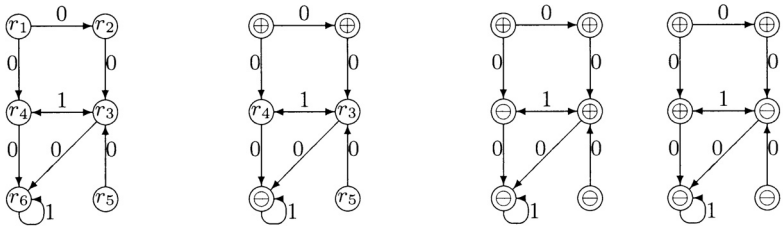


Fig. 1. (a) The *RDG* of logic program Π_1; **(b)** The (partially) colored *RDG* (Γ_{Π_1}, C_2); **(c+d)** The totally colored *RDG*s $(\Gamma_{\Pi_1}, C_{4_a})$ and $(\Gamma_{\Pi_1}, C_{4_b})$.

vertices colored by C with $\oplus$ or $\ominus$. If C is total, $(C_\oplus, C_\ominus)$ is a binary partition of Π. That is, $\Pi = C_\oplus \cup C_\ominus$ and $C_\oplus \cap C_\ominus = \emptyset$. Accordingly, we often identify a coloring C with the pair $(C_\oplus, C_\ominus)$. A *partial* coloring C induces a pair $(C_\oplus, C_\ominus)$ of sets such that $C_\oplus \cup C_\ominus \subseteq \Pi$ and $C_\oplus \cap C_\ominus = \emptyset$. For comparing partial colorings, C and C', we define $C \sqsubseteq C'$, if $C_\oplus \subseteq C'_\oplus$ and $C_\ominus \subseteq C'_\ominus$. The "empty" coloring $(\emptyset, \emptyset)$ is the $\sqsubseteq$-smallest coloring. Accordingly, we define $C \sqcup C'$ as $(C_\oplus \cup C'_\oplus, C_\ominus \cup C'_\ominus)$.

If C is a coloring of Γ_Π, we call the pair (Γ_Π, C) a *colored RDG*. For example, "coloring" the *RDG* of Π_1 with

$$C_2 = (\{r_1, r_2\},\ \{r_6\}) \tag{2}$$

yields the colored graph in Figure 1b. For simplicity, when coloring, we replace the label of a node by the respective color.

The central question addressed in this paper is how to compute the total colorings of *RDG*s that correspond to the answer sets of an underlying program. In fact, the colorings of interest can be distinguished in a straightforward way. Let Π be a logic program along with its *RDG* Γ. Then, for every answer set X of Π, define an *admissible coloring* C of Γ as $C = (R_\Pi(X), \Pi \setminus R_\Pi(X))$. By way of the respective generating rules, we associate with any program a set of admissible colorings whose members are in one-to-one correspondence with its answer sets. Clearly, any admissible coloring is total; also, we have $X = head(C_\oplus)$. We use $AC(\Pi)$ for denoting the set of all admissible colorings of a *RDG* Γ_Π. For a partial coloring C, we define $AC_\Pi(C)$ as the set of all admissible colorings of Γ_Π compatible with C. Formally, given the *RDG* Γ of a logic program Π and a partial coloring C of Γ, define $AC_\Pi(C) = \{C' \in AC(\Pi) \mid C \sqsubseteq C'\}$. Clearly, $C_1 \sqsubseteq C_2$ implies $AC_\Pi(C_1) \supseteq AC_\Pi(C_2)$. Also, note that $AC(\Pi) = AC_\Pi((\emptyset, \emptyset))$. Regarding program Π_1 and coloring C_2, we get $AC_{\Pi_1}(C_2) = AC(\Pi_1) = \{(\{r_1, r_2, r_3\}, \{r_4, r_5, r_6\}), (\{r_1, r_2, r_4\}, \{r_3, r_5, r_6\})\}$ as shown in Figure 1c+d. Accordingly, define $AS_\Pi(C)$ as the set of all answer sets X of Π compatible with partial coloring C: $AS_\Pi(C) = \{X \in AS(\Pi) \mid C_\oplus \subseteq R_\Pi(X)$ and $C_\ominus \cap R_\Pi(X) = \emptyset\}$. As regards program Π_1 and coloring C_2, we get $AS_{\Pi_1}(C_2) = AS(\Pi_1) = \{\{b, p, f\}, \{b, p, f'\}\}$.

We need the following concepts for describing a rule's status of applicability.

Definition 2. *Let $\Gamma = (\Pi, E_0, E_1)$ be the RDG of logic program Π and C be a partial coloring of Γ. For $r \in \Pi$, we define:*

1. r *is supported in* (Γ, C), *if* $body^+(r) \subseteq \{head(r') \mid (r', r) \in E_0, r' \in C_\oplus\}$;
2. r *is unsupported in* (Γ, C), *if* $\{r' \mid (r', r) \in E_0, head(r') = q\} \subseteq C_\ominus$ *for some* $q \in body^+(r)$;
3. r *is blocked in* (Γ, C), *if* $r' \in C_\oplus$ *for some* $(r', r) \in E_1$;
4. r *is unblocked in* (Γ, C), *if* $r' \in C_\ominus$ *for all* $(r', r) \in E_1$.

In what follows, we use $S(\Gamma, C), \overline{S}(\Gamma, C), B(\Gamma, C)$, and $\overline{B}(\Gamma, C)$ for denoting the sets of all supported, unsupported, blocked, and unblocked rules in (Γ, C). For illustration, consider the sets obtained regarding the colored *RDG* (Γ_{Π_1}, C_2) in Figure 1b.

$$\begin{array}{ll} S(\Gamma_{\Pi_1}, C_2) = \{r_1, r_2, r_3, r_4\} & \overline{S}(\Gamma_{\Pi_1}, C_2) = \{r_5\} \\ B(\Gamma_{\Pi_1}, C_2) = \emptyset & \overline{B}(\Gamma_{\Pi_1}, C_2) = \{r_1, r_2, r_5, r_6\} \end{array} \tag{3}$$

The next results are important for understanding the idea of our approach.

Theorem 1. *Let* Γ *be the RDG of logic program* Π *and* C *be a partial coloring of* Γ. *Then, we have for every* $X \in AS_\Pi(C)$ *that*

1. $S(\Gamma, C) \cap \overline{B}(\Gamma, C) \subseteq R_\Pi(X)$;
2. $\overline{S}(\Gamma, C) \cup B(\Gamma, C) \subseteq \Pi \setminus R_\Pi(X)$.

If C *is admissible, we have for* $\{X\} = AS_\Pi(C)$ *that*

3. $S(\Gamma, C) \cap \overline{B}(\Gamma, C) = R_\Pi(X)$;
4. $\overline{S}(\Gamma, C) \cup B(\Gamma, C) = \Pi \setminus R_\Pi(X)$.

Equation 3 and 4 are equivalent since C is total. Reconsider the partially colored *RDG* (Γ_{Π_1}, C_2) in Figure 1b. For every $X \in AS_{\Pi_1}(C_2) = \{\{b, p, f\}, \{b, p, f'\}\}$, we have

$$\begin{array}{ll} S(\Gamma_{\Pi_1}, C_2) \cap \overline{B}(\Gamma_{\Pi_1}, C_2) = \{r_1, r_2\} & \subseteq R_{\Pi_1}(X); \\ \overline{S}(\Gamma_{\Pi_1}, C_2) \cup B(\Gamma_{\Pi_1}, C_2) = \{r_5\} & \subseteq \Pi \setminus R_{\Pi_1}(X). \end{array}$$

3 Deciding Answersetship from Colored Graphs

The result in Theorem 1 started from an existing answer set induced from a given coloring. We now develop concepts that allow us to decide whether a (total) coloring represents an answer set by purely graph-theoretical means. To begin with, we define a graph structure accounting for the notion of recursive support.

Definition 3. *Let* Γ *be the RDG of logic program* Π *and* C *be a partial coloring of* Γ. *We define a support graph of* (Γ, C) *as an acyclic* 0*-subgraph* (V, E) *of* Γ *such that* $body^+(r) \subseteq \{head(r') \mid (r', r) \in E\}$ *for all* $r \in V$, $C_\oplus \subseteq V$, *and* $C_\ominus \cap V = \emptyset$.

Intuitively, support graphs constitute the graph-theoretical counterpart of operator *Cn*. Every uncolored *RDG* (with $C = (\emptyset, \emptyset)$) has a unique support graph possessing a largest set of vertices. We refer to such support graphs as *maximal* ones; all of them share the same set of vertices. For example, the maximal support graph of $(\Gamma_{\Pi_1}, (\emptyset, \emptyset))$, given in Figure 1a, excludes r_5, since it cannot be supported (recursively); otherwise, it contains,

except for (r_5, r_3), all 0-edges of Γ_{Π_1}. The maximal support graph of the colored *RDG* (Γ_{Π_1}, C_2), given in Figure 1b, is $(\{r_1, r_2, r_3, r_4\}, \{(r_1, r_2), (r_1, r_4), (r_2, r_3)\})$. It includes all positively colored and excludes all negatively colored nodes in (Γ_{Π_1}, C_2).

Given a program $\{q, p \leftarrow q\}$ a "bad" coloring, like $C = (\{p \leftarrow q\}, \{q\})$, may deny the existence of a support graph of (Γ, C). As above, we distinguish maximal support graphs of colored graphs through their maximal set of vertices. For colored graphs, we have the following conditions guaranteeing the existence of (maximal) support graphs.

Theorem 2. *Let Γ be the RDG of logic program Π and C be a partial coloring of Γ. If $AC_\Pi(C) \neq \emptyset$, then there is a (maximal) support graph of (Γ, C).*

Clearly, the existence of a support graph implies that of a maximal one. Note furthermore that support graphs of totally colored graphs are necessarily maximal.

Corollary 1. *Let Γ be the RDG of logic program Π and C be an admissible coloring of Γ. Then, $(C_\oplus, E)$ is a (maximal) support graph of (Γ, C) for some $E \subseteq (\Pi \times \Pi)$.*

Taking the last result together with Property 3 or 4 in Theorem 1, we obtain a sufficient characterization of admissible colorings (along with their underlying answer sets).

Theorem 3. *Let Γ be the RDG of logic program Π and let C be a total coloring of Γ. Then, the following statements are equivalent.*

1. *C is an admissible coloring of Γ;*
2. *$C_\oplus = S(\Gamma, C) \cap \overline{B}(\Gamma, C)$ and there is a support graph of (Γ, C);*
3. *$C_\ominus = \overline{S}(\Gamma, C) \cup B(\Gamma, C)$ and there is a support graph of (Γ, C).*

For illustration, let us consider the two admissible colorings of *RDG* Γ_{Π_1}, corresponding to the two answer sets of program Π_1:

$$C_{4_a} = (\{r_1, r_2, r_3\},\ \{r_4, r_5, r_6\}) \quad \text{and} \quad C_{4_b} = (\{r_1, r_2, r_4\},\ \{r_3, r_5, r_6\}). \qquad (4)$$

The resulting colored *RDG*s are depicted in Figure 1c+d. Let us detail the case of C_{4_a}:

$$S(\Gamma_{\Pi_1}, C_{4_a}) \cap \overline{B}(\Gamma_{\Pi_1}, C_{4_a}) = \{r_1, r_2, r_3\} = (C_{4_a})_\oplus;$$
$$\overline{S}(\Gamma_{\Pi_1}, C_{4_a}) \cup B(\Gamma_{\Pi_1}, C_{4_a}) = \{r_4, r_5, r_6\} = (C_{4_a})_\ominus.$$

The maximal support graph of $(\Gamma_{\Pi_1}, C_{4_a})$ is given by $((C_{4_a})_\oplus, \{(r_1, r_2), (r_2, r_3)\})$.

In the full paper, we show how our graph-theoretical approach allows for capturing the original concepts like Cn and Π^X. Also, we introduce the concept of a *blockage graph* by means of 1-subgraphs for capturing blockage relations.

4 Operational Characterizations

The goal of this section is to provide operational characterizations of answer sets. The idea is to start with the empty coloring $(\emptyset, \emptyset)$ and to successively apply operators that turn a partial coloring C into another one C' such that $C \sqsubseteq C'$, if possible. [1] This is done until an admissible coloring, encompassing an answer set, is obtained.

We concentrate first on operations deterministically extending partial colorings.

[1] Recall that $C \sqsubseteq C'$ implies $AC_\Pi(C) \supseteq AC_\Pi(C')$).

Definition 4. *Let Γ be the RDG of logic program Π and C be a partial coloring of Γ. Then, define $\mathcal{P}_\Gamma : \mathbb{C} \to \mathbb{C}$ as $\mathcal{P}_\Gamma(C) = C \sqcup (S(\Gamma, C) \cap \overline{B}(\Gamma, C),\ \overline{S}(\Gamma, C) \cup B(\Gamma, C))$.*

A partial coloring C is closed under $\mathcal{P}_\Gamma$, if $C = \mathcal{P}_\Gamma(C)$. Note that $\mathcal{P}_\Gamma(\ldots)$ does not always exist. To see this, observe that $\mathcal{P}_\Gamma((\{a \leftarrow not\ a\}, \emptyset))$ would be $(\{a \leftarrow not\ a\}, \{a \leftarrow not\ a\})$, which is no mapping and thus no partial coloring.

Interestingly, $\mathcal{P}_\Gamma$ exists on colorings expressing answer sets.

Theorem 4. *Let Γ be the RDG of logic program Π and C a partial coloring of Γ. If $AC_\Pi(C) \neq \emptyset$, then $\mathcal{P}_\Gamma(C)$ exists.*

Note that $\mathcal{P}_\Gamma(C)$ may exist although $AC_\Pi(C) = \emptyset$. To see this, consider $\Pi = \{a \leftarrow , c \leftarrow a, not\ c\}$. Clearly, $AC_\Pi(C) = \emptyset$. However, $\mathcal{P}_\Gamma((\emptyset, \emptyset)) = (\{r_1\}, \emptyset)$ exists.

Now, we can define our principal propagation operator in the following way.

Definition 5. *Let Γ be the RDG of logic program Π and C a partial coloring of Γ. Then, define $\mathcal{P}^*_\Gamma : \mathbb{C} \to \mathbb{C}$ where $\mathcal{P}^*_\Gamma(C)$ is the $\sqsubseteq$-smallest partial coloring containing C and being closed under $\mathcal{P}_\Gamma$.*

An iterative definition of $\mathcal{P}^*_\Gamma$ in terms of $\mathcal{P}_\Gamma$ is given in the full paper.

Like $\mathcal{P}_\Gamma(C)$, $\mathcal{P}^*_\Gamma(C)$ is not necessarily defined. This situation is made precise next.

Theorem 5. *Let Γ be the RDG of logic program Π and C a partial coloring of Γ. If $AC_\Pi(C) \neq \emptyset$, then $\mathcal{P}^*_\Gamma(C)$ exists.*

The non-existence of $\mathcal{P}^*_\Gamma$ is an important feature since an undefined application of $\mathcal{P}^*_\Gamma$ amounts to a backtracking situation at the implementation level. Note that $\mathcal{P}^*_\Gamma((\emptyset, \emptyset))$ always exists, even though we may have $AC_\Pi((\emptyset, \emptyset)) = \emptyset$ (because of $AS(\Pi) = \emptyset$).

For illustration, consider program Π_1. We get:

$$\begin{aligned}
\mathcal{P}_\Gamma((\emptyset, \emptyset)) &= (\{r_1\}, \{r_5\}) \\
\mathcal{P}_\Gamma((\{r_1\}, \{r_5\})) &= (\{r_1, r_2\}, \{r_5\}) \\
\mathcal{P}_\Gamma((\{r_1, r_2\}, \{r_5\})) &= (\{r_1, r_2\}, \{r_5\}) \text{ and so } \mathcal{P}^*_\Gamma((\emptyset, \emptyset)) = (\{r_1, r_2\}, \{r_5\}) .
\end{aligned}$$

Let us now elaborate upon the formal properties of $\mathcal{P}_\Gamma$ and $\mathcal{P}^*_\Gamma$. First, we observe that both are reflexive, that is, $C \sqsubseteq \mathcal{P}_\Gamma(C)$ and $C \sqsubseteq \mathcal{P}^*_\Gamma(C)$ provided they exist. As shown in the full paper, both operators are monotonic: For partial colorings C, C' of Γ such that $AC_\Pi(C') \neq \emptyset$, we have: If $C \sqsubseteq C'$, then $\mathcal{P}_\Gamma(C) \sqsubseteq \mathcal{P}_\Gamma(C')$; analogously for $\mathcal{P}^*_\Gamma$. Consequently, we have $C \sqsubseteq \mathcal{P}_\Gamma(C) \sqsubseteq \mathcal{P}_\Gamma(\mathcal{P}_\Gamma(C))$. Moreover, $\mathcal{P}_\Gamma$ and $\mathcal{P}^*_\Gamma$ are answer set preserving: $AC_\Pi(C) = AC_\Pi(\mathcal{P}_\Gamma(C)) = AC_\Pi(\mathcal{P}^*_\Gamma(C))$.

In fact, $\mathcal{P}_\Gamma$ can be used for deciding answersetship in the following way.

Corollary 2. *Let Γ be the RDG of logic program Π and let C be a total coloring of Γ. Then, C is an admissible coloring of Γ iff $\mathcal{P}_\Gamma(C) = C$ and (Γ, C) has a support graph.*

For relating $\mathcal{P}^*_\Gamma$ to the well-known Fitting operator [7], we need the following.

Definition 6. *Let Γ be the RDG of logic program Π and let C be a partial coloring of Γ. Define $X_C = \{head(r) \mid r \in C_\oplus\}$ and $Y_C = \{q \mid$ for all $r \in \Pi$, if $head(r) = q$, then $r \in C_\ominus\}$.*

The pair (X_C, Y_C) is a 3-valued interpretation of Π. By letting the pair mapping $\Phi_\Pi(X, Y)$ be Fitting's operator [7], we have the following result.

Theorem 6. *Let Γ be the RDG of logic program Π. If $C = \mathcal{P}^*_\Gamma((\emptyset, \emptyset))$, then $\Phi^\omega_\Pi(\emptyset, \emptyset) = (X_C, Y_C)$.*

The next operation draws upon the maximal support graph of colored *RDG*s.

Definition 7. *Let Γ be the RDG of logic program Π and C be a partial coloring of Γ. Furthermore, let (V, E) be a maximal support graph of (Γ, C) for some $E \subseteq (\Pi \times \Pi)$.*

Then, define $\mathcal{U}_\Gamma : \mathbb{C} \to \mathbb{C}$ as $\mathcal{U}_\Gamma(C) = (C_\oplus, \Pi \setminus V)$.

This operator allows for coloring rules with $\ominus$ whenever it is clear from the given partial coloring that they will remain unsupported. Observe that $\Pi \setminus V = C_\ominus \cup (\Pi \setminus V)$. Like $\mathcal{P}^*_\Gamma$, $\mathcal{U}_\Gamma(C)$ is an extension of C. Unlike $\mathcal{P}^*_\Gamma$, however, $\mathcal{U}_\Gamma$ allows for coloring nodes unconnected with the already colored part of the graph. For Π_1, for instance, we obtain $\mathcal{U}_\Gamma((\emptyset, \emptyset)) = (\emptyset, \{r_5\})$. While this information on r_5 can also be supplied by $\mathcal{P}_\Gamma$, it is not obtainable for "self-supporting 0-loops", as in $\Pi = \{p \leftarrow q, q \leftarrow p\}$. In this case, we obtain $\mathcal{U}_\Gamma((\emptyset, \emptyset)) = (\emptyset, \{p \leftarrow q, q \leftarrow p\})$, which is not obtainable through $\mathcal{P}_\Gamma$.

Operator $\mathcal{U}_\Gamma$ is defined on colorings guaranteeing the existence of support graphs.

Corollary 3. *Let Γ be the RDG of logic program Π and C be a partial coloring of Γ.*

If (Γ, C) has a support graph, then $\mathcal{U}_\Gamma(C)$ exists.

We show in the full paper that $\mathcal{U}_\Gamma$ is reflexive, idempotent, monotonic, and answer set preserving. That is, for partial colorings C and C' of Γ such that $AC_\Pi(C) \neq \emptyset$ and $AC_\Pi(C') \neq \emptyset$, we have $C \sqsubseteq \mathcal{U}_\Gamma(C)$, $\mathcal{U}_\Gamma(C) = \mathcal{U}_\Gamma(\mathcal{U}_\Gamma(C))$, and if $C \sqsubseteq C'$, then $\mathcal{U}_\Gamma(C) \sqsubseteq \mathcal{U}_\Gamma(C')$. Moreover, we have $AC_\Pi(C) = AC_\Pi(\mathcal{U}_\Gamma(C))$. Note that unlike $\mathcal{P}_\Gamma$, operator $\mathcal{U}_\Gamma$ leaves the support graph of (Γ, C) unaffected.

Because $\mathcal{U}_\Gamma$ implicitly enforces the existence of a support graph, our operators furnish yet another characterization of answer sets.

Corollary 4. *Let Γ be the RDG of logic program Π and let C be a total coloring of Γ.*

Then, C is an admissible coloring of Γ iff $C = \mathcal{P}_\Gamma(C)$ and $C = \mathcal{U}_\Gamma(C)$.

Note that the last condition cannot guarantee that all supported unblocked rules belong to $C_\oplus$. For instance, $(\emptyset, \{a \leftarrow\})$ has an empty support graph; hence $(\emptyset, \{a \leftarrow\}) = \mathcal{U}_\Gamma((\emptyset, \{a \leftarrow\}))$. That is, the trivially supported fact $a \leftarrow$ remains in $C_\ominus$. In our setting, such a miscoloring is detected by $\mathcal{P}_\Gamma$. That is, $\mathcal{P}_\Gamma((\emptyset, \{a \leftarrow\}))$ is no partial coloring.

Finally, we can express well-founded semantics [17] with our operators. For this, given a partial coloring C, define $(\mathcal{PU})^*_\Gamma(C)$ as the $\sqsubseteq$-smallest partial coloring containing C and being closed under $\mathcal{P}_\Gamma$ and $\mathcal{U}_\Gamma$.

Theorem 7. *Let Γ be the RDG of logic program Π.*

If $C = (\mathcal{PU})^*_\Gamma((\emptyset, \emptyset))$, then (X_C, Y_C) is the well-founded model of Π.

We continue by providing a very general operational characterization that possesses a maximum degree of freedom. To this end, we observe that Corollary 4 can serve as a straightforward *check* for deciding whether a given total coloring constitutes an answer set. A corresponding *guess* can be provided through an operator capturing a non-deterministic (don't know) choice.

Definition 8. *Let Γ be the RDG of logic program Π and C be a partial coloring of Γ. For $\circ \in \{\oplus, \ominus\}$, define $\mathcal{C}^{\circ}_{\Gamma} : \mathbb{C} \to \mathbb{C}$ as*

1. $\mathcal{C}^{\oplus}_{\Gamma}(C) = (C_{\oplus} \cup \{r\}, C_{\ominus})$ *for some* $r \in \Pi \setminus (C_{\oplus} \cup C_{\ominus})$;
2. $\mathcal{C}^{\ominus}_{\Gamma}(C) = (C_{\oplus}, C_{\ominus} \cup \{r\})$ *for some* $r \in \Pi \setminus (C_{\oplus} \cup C_{\ominus})$.

We use $\mathcal{C}^{\circ}_{\Gamma}$ if the distinction between $\mathcal{C}^{\oplus}_{\Gamma}(C)$ and $\mathcal{C}^{\ominus}_{\Gamma}(C)$ is of no importance. Strictly speaking, $\mathcal{C}^{\circ}_{\Gamma}$ is also parametrized with r; we leave this implicit.

Combining the previous guess and check operators yields our first operational characterization of admissible colorings (along with its underlying answer sets).

Theorem 8. *Let Γ be the RDG of logic program Π and let C be a total coloring of Γ. Then, C is an admissible coloring of Γ iff there exists a sequence $(C^i)_{0 \leq i \leq n}$ where*

1. $C^0 = (\emptyset, \emptyset)$;
2. $C^{i+1} = \mathcal{C}^{\circ}_{\Gamma}(C^i)$ *for some* $\circ \in \{\oplus, \ominus\}$ *and* $0 \leq i < n$;
3. $C^n = \mathcal{U}_{\Gamma}(C^n)$;
4. $C^n = \mathcal{P}_{\Gamma}(C^n)$;
5. $C^n = C$.

We refer to such sequences also as *coloring sequences*. Note that all sequences satisfying conditions 1-5 of Theorem 8 are *successful* insofar as their last element corresponds to an existing answer set. If a program has no answer set, then no such sequence exists.

Although this guess and check approach is of no great implementational value, it supplies us with a skeleton for the coloring process that we refine in the sequel. In particular, it stresses the basic fact that we possess complete freedom in forming a coloring sequence as long as we can guarantee that the resulting coloring is a fixed point of $\mathcal{P}_{\Gamma}$ and $\mathcal{U}_{\Gamma}$. It is worth mentioning that this simple approach is inapplicable when fixing $\circ$ to either $\oplus$ or $\ominus$ (see full paper). We observe the following properties.

Theorem 9. *Given the prerequisites in Theorem 8, let $(C^i)_{0 \leq i \leq n}$ be a sequence satisfying conditions 1-5 in Theorem 8. Then, we have the following properties for $0 \leq i \leq n$.*

1. C^i *is a partial coloring;*
2. $C^i \sqsubseteq C^{i+1}$;
3. $AC_{\Pi}(C^i) \supseteq AC_{\Pi}(C^{i+1})$;
4. $AC_{\Pi}(C^i) \neq \emptyset$;
5. (Γ, C^i) *has a (maximal) support graph.*

All these properties represent invariants of the consecutive colorings. While the first three properties are provided by operator $\mathcal{C}^{\circ}_{\Gamma}$ in choosing among uncolored rules only, the last two properties are actually enforced by the "check" on the final coloring C^n expressed by conditions 3–5. In fact, sequences only enjoying conditions 1 and 2 in Theorem 8, fail to satisfy Property 4 and 5. In practical terms, this means that computations of successful sequences may be led numerous times on the "garden path".

As well-known, the number of choices can be significantly reduced by applying deterministic operators.

Theorem 10. *Let Γ be the RDG of logic program Π and let C be a total coloring of Γ. Then, C is an admissible coloring of Γ iff there exists a sequence $(C^i)_{0 \leq i \leq n}$ where*

1. $C^0 = (\mathcal{PU})^*_\Gamma((\emptyset, \emptyset))$;
2. $C^{i+1} = (\mathcal{PU})^*_\Gamma(\mathcal{C}^\circ_\Gamma(C^i))$ *for some* $\circ \in \{\oplus, \ominus\}$ *and* $0 \leq i < n$;
3. $C^n = C$.

The continuous applications of $\mathcal{P}_\Gamma$ and $\mathcal{U}_\Gamma$ extend colorings after each choice. Furthermore, this proceeding guarantees that each partial coloring C^i is closed under $\mathcal{P}_\Gamma$ and $\mathcal{U}_\Gamma$. It is clear in view of Theorem 8 that any number of iterations of $\mathcal{P}_\Gamma$ and $\mathcal{U}_\Gamma$ can be executed after $\mathcal{C}^\circ_\Gamma$ as long as $(\mathcal{PU})^*_\Gamma$ is the final operation leading to C^n in Theorem 10.

For illustration, consider the coloring sequence in Figure 2, obtained for answer set $\{b, p, f'\}$ of program Π_1. The decisive operation in this sequence is the application of $\mathcal{C}^\oplus_\Gamma$ leading to $C(r_3) = \oplus$. The same final result is obtained when choosing $\mathcal{C}^\ominus_\Gamma$ such

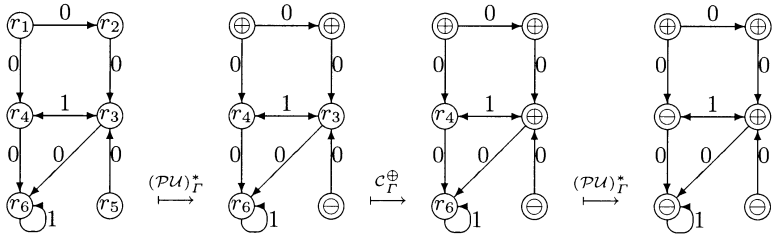


Fig. 2. A coloring sequence.

that $C(r_4) = \ominus$. So, several coloring sequences may lead to the same answer set.

The usage of continuous propagations leads to further invariant properties.

Theorem 11. *Given the prerequisites in Theorem 10, let* $(C^i)_{0 \leq i \leq n}$ *be a sequence satisfying conditions 1-3 in Theorem 10. Then, we have properties 1–5 in Theorem 9 and*

6. $C^{i+1}_\oplus \supseteq S(\Gamma, C^i) \cap \overline{B}(\Gamma, C^i)$;
7. $C^{i+1}_\ominus \supseteq \overline{S}(\Gamma, C^i) \cup B(\Gamma, C^i)$.

Taking Property 6 and 7 together with 5 from Theorem 9, we see that propagation gradually enforces exactly the attributes on partial colorings, expressed in Theorem 3.

Given that we obtain only two additional properties, one may wonder whether exhaustive propagation truly pays off. In fact, its value becomes apparent when looking at the properties of prefix sequences, not necessarily leading to a successful end.

Theorem 12. *Given the prerequisites in Theorem 10, let* $(C^j)_{0 \leq j \leq m}$ *be a sequence satisfying Condition 1 and 2 in Theorem 10.*

Then, we have properties 1–3, 5 in Theorem 9 and 6–7 in Theorem 11.

Using exhaustive propagations, we observe that except for Property 4 all properties of successful sequences, are shared by (possibly unsuccessful) prefix sequences. In the full paper, we prove that propagation leads to shorter and fewer (prefix) sequences.

What else may cut down the number of choices? Looking at the graph structures underlying an admissible coloring, we observe that support graphs possess a non-local,

since recursive, structure, while blockage exhibits a rather local structure, based on arc-wise constraints. Consequently, it seems advisable to prefer choices maintaining support structures over those maintaining blockage relations, since the former have more global repercussions than the latter. To this end, we develop in what follows a strategy that is based on a choice operation restricted to supported rules.

Definition 9. *Let Γ be the RDG of logic program Π and C be a partial coloring of Γ. For $\circ \in \{\oplus, \ominus\}$, define $\mathcal{D}^{\circ}_{\Gamma} : \mathbb{C} \to \mathbb{C}$ as*

1. $\mathcal{D}^{\oplus}_{\Gamma}(C) = (C_{\oplus} \cup \{r\}, C_{\ominus})$ *for some* $r \in S(\Gamma, C) \setminus (C_{\oplus} \cup C_{\ominus})$;
2. $\mathcal{D}^{\ominus}_{\Gamma}(C) = (C_{\oplus}, C_{\ominus} \cup \{r\})$ *for some* $r \in S(\Gamma, C) \setminus (C_{\oplus} \cup C_{\ominus})$.

The number of rules colorable by $\mathcal{D}^{\circ}_{\Gamma}$ is normally smaller than that by $\mathcal{C}^{\circ}_{\Gamma}$. Depending on how the non-determinism of $\mathcal{D}^{\circ}_{\Gamma}$ is dealt with algorithmically, this may either lead to a reduced depth of the search tree or a reduced branching factor.

In a successful coloring sequence $(C^i)_{0 \leq i \leq n}$, all rules in $C^n_{\oplus}$ belong to an encompassing support graph. Furthermore, using $\mathcal{D}^{\oplus}_{\Gamma}(C)$ (and $\mathcal{P}^{*}_{\Gamma}$) the supportness of each set $C^i_{\oplus}$ is made invariant. Consequently, such a proceeding allows for establishing the existence of support graphs "on the fly" and offers a much simpler approach to the task(s) previously accomplished by $\mathcal{U}_{\Gamma}$. In fact, one may completely dispose of operator $\mathcal{U}_{\Gamma}$ and color in a final step all uncolored rules with $\ominus$.

Definition 10. *Let Γ be the RDG of logic program Π and C a partial coloring of Γ. Then, define $\mathcal{N}_{\Gamma} : \mathbb{C} \to \mathbb{C}$ as $\mathcal{N}_{\Gamma}(C) = (C_{\oplus}, \Pi \setminus C_{\oplus})$.*

Roughly speaking, the idea is then to "actively" color only supported rules and rules blocked by supported rules; all remaining rules are then unsupported and "thrown" into $C_{\ominus}$ in a final step.

Theorem 13. *Let Γ be the RDG of logic program Π and let C be a total coloring of Γ. Then, C is an admissible coloring of Γ iff there exists a sequence $(C^i)_{0 \leq i \leq n}$ where*

1. $C^0 = (\emptyset, \emptyset)$;
2. $C^{i+1} = \mathcal{D}^{\circ}_{\Gamma}(C^i)$ *where* $\circ \in \{\oplus, \ominus\}$ *and* $0 \leq i < n-1$;
3. $C^n = \mathcal{N}_{\Gamma}(C^{n-1})$;
4. $C^n = \mathcal{P}_{\Gamma}(C^n)$;
5. $C^n = C$.

We note that there is a little price to pay for turning $\mathcal{U}_{\Gamma}$ into $\mathcal{N}_{\Gamma}$, expressed in Condition 4. Without it, one could use $\mathcal{N}_{\Gamma}$ to obtain a total coloring by coloring rules with $\ominus$ in an arbitrary way. We obtain the following properties for this type of sequences.

Theorem 14. *Given the prerequisites in Theorem 13, let $(C^i)_{0 \leq i \leq n}$ be a sequence satisfying conditions 1-5 in Theorem 13. Then, we have properties 1–5 in Theorem 9 and*

8. *$(C^i_{\oplus}, E)$ is a support graph of (Γ, C^i) for some $E \subseteq \Pi \times \Pi$.*

Unlike the coloring sequences only enjoying Condition 5 in Theorem 9, the sequences formed by means of $\mathcal{D}^{\circ}_{\Gamma}$ guarantee that each $C^i_{\oplus}$ forms an independent support graph.

In fact, there is some overlap among operator $\mathcal{D}_\Gamma^\ominus$ and $\mathcal{N}_\Gamma$. To see this, consider $\Pi = \{a \leftarrow , b \leftarrow not\ a\}$. Initially, we must apply $\mathcal{D}_\Gamma^\oplus$ to obtain $(\{a \leftarrow\}, \emptyset)$ from $(\emptyset, \emptyset)$. Then, however, we may either apply $\mathcal{D}_\Gamma^\ominus$ or $\mathcal{N}_\Gamma$ for obtaining admissible coloring $(\{a\}, \{b \leftarrow not\ a\})$. Interestingly, this overlap can be eliminated by adding propagation operator $\mathcal{P}_\Gamma^*$. This results in the basic strategy used in the `noMoRe` system [1].

Theorem 15. *Let Γ be the RDG of logic program Π and let C be a total coloring of Γ. Then, C is an admissible coloring of Γ iff there exists a sequence $(C^i)_{0 \leq i \leq n}$ where*

1. $C^0 = \mathcal{P}_\Gamma^*((\emptyset, \emptyset))$;
2. $C^{i+1} = \mathcal{P}_\Gamma^*(\mathcal{D}_\Gamma^\circ(C^i))$ *where* $\circ \in \{\oplus, \ominus\}$ *and* $0 \leq i < n-1$;
3. $C^n = \mathcal{N}_\Gamma(C^{n-1})$;
4. $C^n = \mathcal{P}_\Gamma(C^n)$;
5. $C^n = C$.

Indeed, the strategy of `noMoRe` applies operator $\mathcal{D}_\Gamma^\circ$ as long as there are supported rules. Once no more uncolored supported rules exist, operator $\mathcal{N}_\Gamma$ is called. Finally, $\mathcal{P}_\Gamma$ is applied (in practice, only to those rules colored previously by $\mathcal{N}_\Gamma$). At first sight, this approach may seem to correspond to a subclass of the coloring sequences described in Theorem 15, in the sense that `noMoRe` enforces a maximum number of transitions described in Condition 2. To see that this is not the case, we observe the following property.

Theorem 16. *Given the prerequisites in Theorem 15, let $(C^i)_{0 \leq i \leq n}$ be a sequence satisfying conditions 1-5 in Theorem 15. Then, we have $(\mathcal{N}_\Gamma(C^{n-1})_\ominus \setminus C_\ominus^{n-1}) \subseteq \overline{S}(\Gamma, C)$.*

That is, no matter which (supported) rules are colored $\ominus$ by $\mathcal{D}_\Gamma^\ominus$, operator $\mathcal{N}_\Gamma$ only applies to unsupported ones. It is thus no restriction to enforce the consecutive application of $\mathcal{P}_\Gamma^*$ and $\mathcal{D}_\Gamma^\circ$ until no more supported rules are available. In fact, it is the interplay of the two last operators that guarantees this property. For instance, looking at $\Pi = \{a, b \leftarrow not\ a\}$, we see that we directly obtain the final total coloring because $(\{a\}, \{b \leftarrow not\ a\}) = \mathcal{P}_\Gamma^*(\mathcal{D}_\Gamma^\oplus((\emptyset, \emptyset)))$, without any appeal to $\mathcal{N}_\Gamma$. Rather it is $\mathcal{P}_\Gamma^*$ that detects that $b \leftarrow not\ a$ is blocked. Generally speaking, $\mathcal{D}_\Gamma^\oplus$ consecutively chooses the generating rules of an answer set, finally gathered in $C_\oplus = S(\Gamma, C) \cap \overline{B}(\Gamma, C)$. Clearly, every rule in $B(\Gamma, C)$ is blocked by some rule in $C_\oplus$. So whenever a rule r is added by $\mathcal{D}_\Gamma^\oplus$ to $C_\oplus$, operator $\mathcal{P}_\Gamma^*$ adds all rules blocked by r to $C_\ominus$. In this way, $\mathcal{P}_\Gamma^*$ and $\mathcal{D}_\Gamma^\oplus$ gradually color all rules in $S(\Gamma, C) \cap \overline{B}(\Gamma, C)$ and $B(\Gamma, C)$, so that all remaining uncolored rules, subsequently treated by $\mathcal{N}_\Gamma$, must belong to $\overline{S}(\Gamma, C)$. We obtain the following properties.

Theorem 17. *Given the prerequisites in Theorem 15, let $(C^i)_{0 \leq i \leq n}$ be a sequence satisfying conditions 1-5 in Theorem 15. Then, we have properties 1–5 in Theorem 9, 6–7 in Theorem 11, and 8 in Theorem 14.*

In the full paper, we discuss alternative support-driven operational characterizations using an incremantal version of $\mathcal{U}_\Gamma$ instead of $\mathcal{N}_\Gamma$. As well, we elaborate upon unicoloring strategies, using only one of the choice operators for $\oplus$ or $\ominus$ instead of both.

5 Discussion, Related Work, and Conclusions

Among the many graph-based approaches in the literature, we find some dealing with stratification [2], existence of answer sets [6,3], or the actual characterization of answer sets or well-founded semantics [4,3,9,11]. Our own approach has its roots in earlier work on default logic [12,13,16]. The usage of rule-oriented dependency graphs is common to [4,3,9]. In fact, the coloration of such graphs for characterizing answer sets was independently developed in [3] and [9]. While we borrow the term of an admissible coloring from the former, the work reported in Section 3 builds upon the latter and revises its definitions by appeal to the concept of a support graph. [2]

Our major goal is however to provide an operational framework for answer set formation that allows us to bridge the gap between formal yet static characterizations of answer sets and algorithms for computing them. For instance, in the seminal paper [15] describing the `smodels` approach, answer sets are given in terms of so-called *full-sets* and their computation is directly expressed in terms of procedural algorithms. Our operational semantics aims at offering an intermediate stage that facilitates the formal elaboration of computational approaches. Our approach is strongly inspired by the concept of a derivation, in particular, that of an SLD-derivation [14]. This attributes our coloring sequences the flavor of a derivation in a family of calculi, whose respective set of inference rules correspond to the selection of operators.

Although we leave out implementational issues, some remarks relating our approach, and thus the resulting `noMoRe` system [1], to the ones underlying `dlv` [5] and `smodels` [15] are in order. A principal difference manifests itself in how choices are performed. While the two latter's choice is based on atoms occurring (negatively) in the underlying program, our choices are based on its rules. An advantage of our approach is that we can guarantee the support of rules on the fly. Unlike this, support checking is a recurring operation in the `smodels` system, similar to operator $\mathcal{U}_\Gamma$. On the other hand, this approach ensures that the `smodels` algorithm runs in linear space complexity, while a graph-based approach needs quadratic space in the worst case. This "investment" pays off once one is able to exploit the additional structural information offered by a graph. First steps in this direction are made in [10], where graph compressions are described that allow for conflating entire subgraphs into single nodes. Propagation is more or less done similarly in all three approaches. `smodels` relies on computing well-founded semantics, whereas `dlv` uses Fitting's operator plus some backpropagation mechanisms.

To sum up, we build upon the basic graph-theoretical characterizations in [9,11] for developing an operational framework for non-deterministic answer set formation. The general idea is to start from an uncolored *RDG* and to employ specific operators that turn a partially colored graph gradually in a totally colored one, representing an answer set. To this end, we have developed a variety of deterministic and non-deterministic operators. Different coloring sequences (enjoying different formal properties) are obtained by selecting different combinations of operators. Among others, we have identified the particular strategy of the `noMoRe` system as well as operations yielding Fitting's and

[2] The definition of *RDG*s differs from that of "block graphs" in [9], whose practically motivated restrictions are superfluous from a theoretical perspective. Also, we abandon the latter term in order to give the same status to support and blockage relations.

well-founded semantics. Taken together, the last results show that noMoRe's principal propagation operation amounts to applying Fitting's operator. Notably, the explicit detection of 0-loops is avoided by employing a support-driven choice operation. The noMoRe system is available at http://www.cs.uni-potsdam.de/~linke/nomore.

Acknowledgements. This work was supported by the German Science Foundation (DFG) under grant SCHA 550/6, TP C.

References

1. C. Anger, K. Konczak, and T. Linke. noMoRe: Non-monotonic reasoning with logic programs. In S. Flesca et al., editors, *Proceedings of the Eighth European Conference on Logics in Artificial Intelligence (JELIA'02)*, pages 521–524. Springer, 2002.
2. K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1987.
3. G. Brignoli, S. Costantini, O. D'Antona, and A. Provetti. Characterizing and computing stable models of logic programs: the non-stratified case. In C. Baral and H. Mohanty, editors, *Proceedings of the Conference on Information Technology, Bhubaneswar, India*, pages 197–201. AAAI Press, 1999.
4. Y. Dimopoulos and A. Torres. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170:209–244, 1996.
5. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for nonmonotonic reasoning. In J. Dix et al., editors, *Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 363–374. Springer, 1997.
6. F. Fages. Consistency of clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
7. M. Fitting. Fixpoint semantics for logic programming a survey. *Theoretical Computer Science*, 278(1-2):25–51, 2002.
8. M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing*, 9:365–385, 1991.
9. T. Linke. Graph theoretical characterization and computation of answer sets. In B. Nebel, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 641–645. Morgan Kaufmann Publishers, 2001.
10. T. Linke. Using nested logic programs for answer set programming. In M. De Voss and A. Provetti, editors, *Proceedings of the Workshop on Answer Set Programming: Advances in Theory and Implementation (ASP03)*. 181–194, CEUR, 2003.
11. T. Linke, C. Anger, and K. Konczak. More on noMoRe. In S. Flesca et al., editors, *Proceedings of the Eighth European Conference on Logics in Artificial Intelligence (JELIA'02)*, pages 468–480, 2002.
12. T. Linke and T. Schaub. An approach to query-answering in Reiter's default logic and the underlying existence of extensions problem. In J. Dix et al., editors, *Proceedings of the Sixth European Workshop on Logics in Artificial Intelligence*, pages 233–247. Springer, 1998.
13. T. Linke and T. Schaub. Alternative foundations for Reiter's default logic. *Artificial Intelligence*, 124(1):31–86, 2000.
14. J. Lloyd. *Foundations of Logic Programming*. Springer, 1987.
15. I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303. The MIT Press, 1996.

16. C. Papadimitriou and M. Sideri. Default theories that always have extensions. *Artificial Intelligence*, 69:347–357, 1994.
17. A. van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Nondefinite vs. Definite Causal Theories

Joohyung Lee

Department of Computer Sciences
University of Texas, Austin, TX, USA
appsmurf@cs.utexas.edu

Abstract. Nonmonotonic causal logic can be used to represent properties of actions, including actions with conditional and indirect effects, nondeterministic actions, and concurrently executed actions. The definite fragment of causal logic can be mapped to propositional logic by the process of completion, and this idea has led to the development of the Causal Calculator. In this note, we show how to turn arbitrary causal theories into definite theories without changing the sets of models. The translation consists of two parts: one is a set of definite rules which is obtained from the given theory by translating each rule one by one, in a modular way, and the other is a set of constraints similar to loop formulas for logic programs. Our result characterizes the semantics of causal logic in terms of propositional logic and tells us that an essential difference between the semantics of causal logic and the answer set semantics is related to the definition of a loop in each.

1 Introduction

The nonmonotonic causal logic defined in [McCain and Truner, 1997, Giunchiglia *et al.*, 2003] can be used to represent properties of actions, including actions with conditional and indirect effects, nondeterministic actions, and concurrently executed actions. Action language $\mathcal{C}+$ [Giunchiglia *et al.*, 2003], a high level notation for causal logic, is a useful formalism for describing transition systems– directed graphs whose vertices correspond to states and whose edges correspond to the executions of actions.

For the "definite" fragment of causal logic and $\mathcal{C}+$, the problem of determining the existence of models can be reduced to the satisfiability problem for propositional logic by the process of completion [McCain and Turner, 1997]–a translation similar to Clark's completion [Clark, 1978] familiar from logic programming. The Causal Calculator (CCALC)[1] is an implementation of the definite fragment of causal logic based on this idea. After turning a definite causal theory into a classical propositional theory, CCALC finds the models of the latter by invoking a satisfiability solver, such as CHAFF[2], SATO[3] and RELSAT[4].

[1] http://www.cs.utexas.edu/users/tag/ccalc/ .
[2] http://www.ee.princeton.edu/~chaff/.
[3] http://www.cs.uiowa.edu/~hzhang/sato.html.
[4] http://www.almaden.ibm.com/cs/people/bayardo/resources.html.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 141–153, 2004.

Although the limitation of definiteness may look too restrictive, CCALC has been successfully applied to several challenge problems in the theory of commonsense knowledge [Lifschitz, 2000], [Lifschitz *et al.*, 2000], [Campbell and Lifschitz, 2003], [Lee and Lifschitz, 2003a], [Akman *et al.*, 2003] and to the formalization of multi-agent computational systems [Artikis *et al.*, 2003a, Artikis *et al.*, 2003b, Chopra and Singh, 2003].

In this note we show that the limitation is indeed not essential: we show how to turn *arbitrary* causal theories into definite theories without changing the sets of models. The translation consists of two parts: one is a set of definite rules which is obtained from the given theory by translating each rule one by one, in a modular way, and the other is a set of constraints. The main idea of the translation is related to the concept of a loop formula introduced in [Lin and Zhao, 2002], and our main theorem is similar to Theorem 1 from [Lee and Lifschitz, 2003b], which generalizes the result of [Lin and Zhao, 2002] to disjunctive logic programs. The theorem in this note tells us the relationship between nondefinite and definite theories and characterizes the semantics of causal logic in terms of propositional logic. Surprisingly, this result, together with Theorem 1 from [Lee and Lifschitz, 2003b] that characterizes the semantics of logic programs also in terms of propositional logic, tells us that an essential difference between the semantics of causal logic and the answer set semantics is related to the definition of a loop in each. This idea can guide us in translating a representation in one formalism to the other. The proposed translation from causal logic to propositional logic is also interesting from a computational point of view because it can be used to make CCALC understand arbitrary causal theories, not necessarily definite.

The problem of determining that a finite causal theory is consistent is Σ_2^P-hard [Giunchiglia *et al.*, 2003, Proposition 3], while the same problem for a definite causal theory is in NP. Not surprisingly, our reduction of arbitrary causal theories to definite theories introduces an exponential number of loop formulas in the worst case.

In the next section, we review the definition of causal logic and give an example of a nondefinite theory that formalizes an action domain. The translation from arbitrary causal theories to definite theories is described in Section 3; the relationship between causal logic and logic programs is discussed in Section 4.

2 Review of Causal Logic

2.1 Definition

Begin with a propositional signature σ.[5] By a *(causal) rule* we mean an expression of the form

$$F \Leftarrow G$$

[5] Formulas defined in [Giunchiglia *et al.*, 2003] are slightly more general than formulas of the usual propositional logic. In this note, we restrict attention to propositional signatures only.

("F is caused if G holds"), where F, G are formulas in propositional logic of the signature σ. F is called the *head* and G is called the *body* of the rule. Rules with the head $\bot$ are called *constraints*.

A *causal theory* is a finite set of causal rules. A causal theory is called *definite* if the head of every rule in it is either a literal or $\bot$.

Let T be a causal theory, and let I be an interpretation of its signature. The *reduct* T^I of T relative to I is the set of the heads of all rules in T whose bodies are satisfied by I. We say that I is a *model* of T if I is the unique model of T^I.

Intuitively, T^I is the set of formulas that are caused, according to the rules of T, under interpretation I. If this set has no models or more than one model, then, according to the definition above, I is not considered a model of T. If T^I has exactly one model, but that model is different from I, then I is not a model of T either. The only case when I is a model of T is when I satisfies every formula in the reduct, and no other interpretation does.

2.2 Examples

In Section 2.3 of [Giunchiglia *et al.*, 2003], the definition of causal logic is illustrated by the following example. Let T_1 be the following causal theory whose signature is $\{p, q\}$:

$$\begin{aligned} p &\Leftarrow q \\ q &\Leftarrow q \\ \neg q &\Leftarrow \neg q. \end{aligned}$$

Consider, one by one, all interpretations of that signature (we identify an interpretation with the set of literals that are true in it):

- $I_1 = \{p, q\}$. The reduct consists of the heads of the first two rules of T_1: $T_1^{I_1} = \{p, q\}$. Since I_1 is the unique model of $T_1^{I_1}$, it is a model of T_1.
- $I_2 = \{\neg p, q\}$. The reduct is the same as above, and I_2 is not a model of the reduct. Consequently, I_2 is not a model of T_1.
- $I_3 = \{p, \neg q\}$. The only element of the reduct is the head of the third rule of T_1: $T_1^{I_3} = \{\neg q\}$. It has two models. Consequently, I_3 is not a model of T_1.
- $I_4 = \{\neg p, \neg q\}$. The reduct is the same as above, so that I_4 is not a model of T_1 either.

We see that I_1 is the only model of T_1.

Consider another example T_2 whose signature is $\{p, q\}$:

$$\begin{aligned} p \vee \neg q &\Leftarrow \top \\ \neg p \vee q &\Leftarrow \top. \end{aligned}$$

T_2^I is equal to the set of the heads of the rules in T_2 regardless of interpretation I; T_2^I has two models, $\{p, q\}$ and $\{\neg p, \neg q\}$, so that T_2 has no models.

T_3 is the following theory of the same signature that adds one rule to T_2:

$$\begin{aligned} p \vee \neg q &\Leftarrow \top \\ \neg p \vee q &\Leftarrow \top \\ p \vee q &\Leftarrow \top. \end{aligned}$$

Similarly to the previous example, T_3^I is equal to the set of the heads of the rules in T_3 regardless of interpretation I. Now $\{\neg p, \neg q\}$ is not a model of T_3^I, so that T_3 has one model: $\{p, q\}$.

Let T_4 be the following causal theory whose signature is $\{p, q, r\}$:

$$\begin{aligned} p \vee \neg q &\Leftarrow \top \\ \neg p \vee q &\Leftarrow \top \\ p \vee r &\Leftarrow \top \\ \neg r &\Leftarrow \neg r. \end{aligned}$$

We see that T_4 has only one model: $\{p, q, \neg r\}$.

In the case when the theory is definite, it is known that its models can be computed by the process of completion. Consider a definite causal theory T of a signature σ. For each literal l, the *completion formula* for l is the formula

$$l \equiv G_1 \vee \cdots \vee G_n$$

where $G_1, \cdots G_n$ $(n \geq 0)$ are the bodies of the rules of T with head l. The *completion* of T is obtained by taking the completion formulas for all literals of σ, along with the formula $\neg F$ for each constraint $\bot \Leftarrow F$ in T.

For example, the completion of T_1 is

$$\begin{aligned} p &\equiv q \\ \neg p &\equiv \bot \\ q &\equiv q \\ \neg q &\equiv \neg q, \end{aligned}$$

and its only model is $\{p, q\}$, which is exactly the model found above using the definition of causal logic. However, the method of completion is not applicable to nondefinite theories, such as T_2, T_3 and T_4. In Section 3.3, we show how completion can be extended to cover nondefinite theories.

We understand the term *clause* to mean a disjunction of distinct literals $l_1 \vee \cdots \vee l_n$ $(n \geq 0)$, and identify a clause with the corresponding set $\{l_1, \ldots, l_n\}$. Using proposition 4 of [Giunchiglia *et al.*, 2003], any causal theory can be rewritten as a theory of the "normal form" without changing the set of models:

Fact 1 *Any rule of a causal theory T can be replaced with a finite set of rules of the form*

$$F \Leftarrow G \tag{1}$$

where F is a clause and G is a formula, without changing the set of models of T.

2.3 An Example of a Nondefinite Causal Theory for Action Domains

There are some cases where nondefinite theories look natural to formalize certain action domains. An action domain of this kind, due to Marc Denecker, is discussed in [McCain, 1997, Section 7.5]:

Imagine that there are two gears, each powered by a separate motor. There are switches that toggle the motors on and off, and there is a button that moves the gears so as to connect or disconnect them from one another. The motors turn the gears in opposite (i.e., compatible) directions. A gear is caused to turn if either its motor is on or it is connected to a gear that is turning.

McCain's formalization of the example in causal logic is reproduced in Figure 1. The signature consists of symbols

$$MotorOn(x),\ Connected,\ Turning(x)$$

prefixed by time stamp i : where $i \in \{0, \ldots, m\}$ and symbols

$$Toggle(x),\ Push$$

prefixed by time stamp i : where $i \in \{0, \ldots, m-1\}$; x ranges over $\{1, 2\}$. In Figure 1, the expression of the form $i{:}\neg c$ where c is an atom stands for a literal $\neg\, i{:}c$. The presence of the last rule makes the theory nondefinite.

$$
\begin{array}{rcl}
i{+}1{:}\,MotorOn(x) & \Leftarrow & i{:}\,Toggle(x) \wedge\ i{:}\neg MotorOn(x) \\
i{+}1{:}\neg MotorOn(x) & \Leftarrow & i{:}\,Toggle(x) \wedge\ i{:}\,MotorOn(x) \\
i{+}1{:}\,Connected & \Leftarrow & i{:}\,Push \wedge\ i{:}\neg Connected \\
i{+}1{:}\neg Connected & \Leftarrow & i{:}\,Push \wedge\ i{:}\,Connected \\
i+1{:}\,MotorOn(x) & \Leftarrow & i+1{:}\,MotorOn(x) \wedge\ i{:}\,MotorOn(x) \\
i+1{:}\neg MotorOn(x) & \Leftarrow & i+1{:}\neg MotorOn(x) \wedge\ i{:}\neg MotorOn(x) \\
i+1{:}\,Connected(x) & \Leftarrow & i+1{:}\,Connected(x) \wedge\ i{:}\,Connected(x) \\
i+1{:}\neg Connected(x) & \Leftarrow & i+1{:}\neg Connected(x) \wedge\ i{:}\neg Connected(x) \\
0{:}\,MotorOn & \Leftarrow & 0{:}\,MotorOn \\
0{:}\neg MotorOn & \Leftarrow & 0{:}\neg MotorOn \\
0{:}\,Connected & \Leftarrow & 0{:}\,Connected \\
0{:}\neg Connected & \Leftarrow & 0{:}\neg Connected \\
i{:}\,Toggle(x) & \Leftarrow & i{:}\,Toggle(x) \\
i{:}\neg Toggle(x) & \Leftarrow & i{:}\neg Toggle(x) \\
i{:}\,Push & \Leftarrow & i{:}\,Push \\
i{:}\neg Push & \Leftarrow & i{:}\neg Push \\
(i = 0, \ldots, m-1) & &
\end{array}
$$

$$
\begin{array}{rcl}
i{:}\,Turning(x) & \Leftarrow & i{:}\,MotorOn(x) \\
i{:}\neg Turning(x) & \Leftarrow & i{:}\neg Turning(x) \\
i{:}\,Turning(1) \equiv\ i{:}\,Turning(2) & \Leftarrow & i{:}\,Connected \\
(i = 0, \ldots, m) & &
\end{array}
$$

Fig. 1. Two Gears

3 Translation to Definite Theories

3.1 Translating the Rules

Let T be a causal theory of a signature σ whose rules have the form (1). For every rule $F \Leftarrow G$ of T, the corresponding definite rule $DR(F \Leftarrow G)$ is defined as follows:

$$DR(F \Leftarrow G) = \left\{ l \Leftarrow G \wedge \bigwedge_{l' \in F \setminus \{l\}} \overline{l'} \ : \ l \in F \right\}^6$$

if $|F| > 1$; $DR(F \Leftarrow G) = \{F \Leftarrow G\}$ otherwise.

A main part of the translation, the set of definite rules corresponding to T, is the union of $DR(r)$ for all rules r in T:

$$DR(T) = \bigcup_{r \in T} DR(r).$$

Note that $DR(T) = T$ when T is definite.

For example, $DR(T_3)$ is

$$\begin{aligned} p &\Leftarrow q \\ \neg q &\Leftarrow \neg p \\ \neg p &\Leftarrow \neg q \\ q &\Leftarrow p \\ p &\Leftarrow \neg q \\ q &\Leftarrow \neg p, \end{aligned}$$

and its only model is $\{p, q\}$, which is the only model of T_3 also.

The following proposition describes a relationship between the two causal theories.

Proposition 1 *For any causal theory T whose rules have the form (1), every model of T is a model of $DR(T)$.*

However, the converse of Proposition 1 does not hold as shown in the following examples. $DR(T_2)$ is

$$\begin{aligned} p &\Leftarrow q \\ \neg q &\Leftarrow \neg p \\ \neg p &\Leftarrow \neg q \\ q &\Leftarrow p, \end{aligned}$$

and it has two models, $\{p, q\}$ and $\{\neg p, \neg q\}$, while T_2 has no models.

Similarly, $DR(T_4)$ is

$$\begin{aligned} p &\Leftarrow q \\ \neg q &\Leftarrow \neg p \\ \neg p &\Leftarrow \neg q \\ q &\Leftarrow p \\ p &\Leftarrow \neg r \\ r &\Leftarrow \neg p \\ \neg r &\Leftarrow \neg r, \end{aligned}$$

[6] $\overline{l}$ denotes the literal complementary to literal l.

and it has two models, $\{p, q, \neg r\}$ and $\{\neg p, \neg q, r\}$, while the first is the only model of T_4.

In the next section we show how to strengthen the corresponding definite theory so that its models are exactly the models of the original theory.

3.2 Loops and Loop Formulas

Let T be a causal theory of a signature σ whose rules have the form (1). The *head dependency graph* of T is the directed graph G such that

- the vertices of G are the literals of σ, and
- for every rule $l_1 \vee \cdots \vee l_n \Leftarrow F$ of T, G has an edge from each l_i to each $\overline{l_j}$ where $j \neq i$, $1 \leq i, j \leq n$.

Notice that the head dependency graph is similar to the positive dependency graph defined in [Lee and Lifschitz, 2003b].

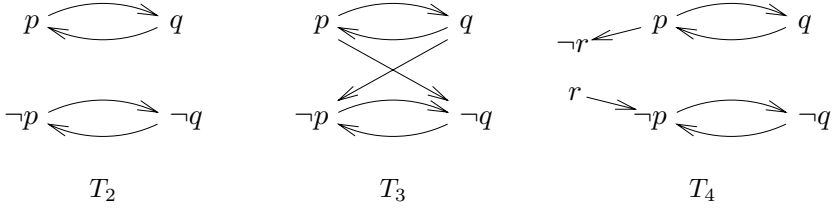


Fig. 2. Head dependency graphs for T_2, T_3, T_4

A nonempty set L of literals is called a *loop* of T if, for every pair l_1, l_2 of literals in L, there exists a path of non-zero length from l_1 to l_2 in the head dependency graph of T such that all vertices in this path belong to L. It is clear that every definite theory, including T_1, has no loops. Each of T_2, T_3 and T_4 has two loops: $\{p, q\}$ and $\{\neg p, \neg q\}$.

Given a set X of literals, $\overline{X}$ is the set of literals complementary to literals in X. The following fact easily follows from the definition.

Fact 2 *For any causal theory T whose rules have the form (1), if a set L of literals is a loop of T, then $\overline{L}$ is a loop of T also.*

For any loop L of T, by $R(L)$ we denote the set of formulas

$$G \wedge \bigwedge_{l \in F \setminus L} \overline{l}$$

for all rules (1) in T such that $F \cap L \neq \emptyset$ and $F \cap \overline{L} = \emptyset$. By $CLF(L)$ we denote the *conjunctive loop formula* of L:

$$CLF(L) = \bigwedge L \supset \bigvee R(L).^{7} \tag{2}$$

By $CLC(T)$ we denote the set of all constraints that express the conjunctive loop formulas for T:

$$CLC(T) = \{\bot \Leftarrow \neg CLF(L) : L \textit{ is a loop of } T\}.$$

For instance,

$$CLC(T_2) = \{\bot \Leftarrow \neg(p \wedge q \supset \bot),\ \bot \Leftarrow \neg(\neg p \wedge \neg q \supset \bot)\},$$
$$CLC(T_3) = \{\bot \Leftarrow \neg(p \wedge q \supset \top),\ \bot \Leftarrow \neg(\neg p \wedge \neg q \supset \bot)\},$$
$$CLC(T_4) = \{\bot \Leftarrow \neg(p \wedge q \supset \neg r),\ \bot \Leftarrow \neg(\neg p \wedge \neg q \supset \bot)\}.$$

In loop formulas of another kind, $\bigwedge$ in the antecedent of (7) is replaced with $\bigvee$. Let's call this formula $DLF(L)$ (*disjunctive loop formula*):

$$DLF(L) = \bigvee L \supset \bigvee R(L). \tag{3}$$

$DLC(T)$ is the set of all constraints that express the disjunctive loop formulas for T.

3.3 Main Theorem

Theorem 1 *For any causal theory T whose rules have the form (1) and any set X of literals, the following conditions are equivalent:*

(a) X is a model of T,
(b) X is a model of $DR(T) \cup CLC(T)$,
(c) X is a model of $DR(T) \cup DLC(T)$.

The theorem tells us that the models of any causal theory T whose rules have the form (1) can be found by computing the models of the completion of the corresponding definite theory $DR(T) \cup CLC(T)$ (or $DR(T) \cup DLC(T)$). Recall that by Fact 1, an arbitrary causal theory can be rewritten as a causal theory whose rules have the form (1) without changing the set of models.

According to Proposition 5 from [Giunchiglia *et al.*, 2003], adding constraints to a causal theory T does not introduce new models, but simply eliminates the models of T that violate at least one of these constraints. That is, adding $\bot \Leftarrow \neg F$ eliminates the models that do not satisfy F. Since $CLC(T)$ break (or $DLC(T)$) is a set of constraints, the models of $DR(T) \cup CLC(T)$ are exactly the models of $DR(T)$ that satisfy every loop formula for T.

[7] The expression $\bigwedge L$ in the antecedent stands for the conjunction of all elements of L. The expression in the consequent has a similar meaning.

For instance, each of the models $\{p, q\}$ and $\{\neg p, \neg q\}$ of $DR(T_2)$ does not satisfy one of the loop formulas $p \wedge q \supset \bot$ and $\neg p \wedge \neg q \supset \bot$, so that T_2 has no models. T_3 has one model because the only model $\{p, q\}$ of $DR(T_3)$ satisfies both loop formulas $p \wedge q \supset \top$ and $\neg p \wedge \neg q \supset \bot$. One of the models $\{p, q, \neg r\}$ of $DR(T_4)$ is also the model of T_4 because it satisfies both loop formulas for T_4; the other model $\{\neg p, \neg q, r\}$ of $DR(T_4)$ is not a model of T_4 because it does not satisfy the loop formula $\neg p \wedge \neg q \supset \bot$.

From the fact that (b) and (c) are equivalent to each other we can infer that the antecedent of a loop formula can be more general. For any loop L, let F_L be a formula formed from literals in L using conjunctions and disjunctions. For instance, F_L can be any single literal that belongs to L or a conjunction or disjunction of several literals from L. It is clear that F_L entails $\bigvee L$ and is entailed by $\bigwedge L$. We conclude:

Corollary 1 *For any causal theory T whose rules have the form (1) and any set X of literals, the condition*

(d) X is a model of $DR(T) \cup \{\bot \Leftarrow \neg(F_L \supset \bigvee R(L)) \ : \ L$ is a loop of $T\}$

is equivalent to each of conditions (a)–(c) from Theorem 1.

The last rule of the description in Figure 1 can be replaced by the following rule without changing the set of models of the theory:

$$\neg \mathit{Turning}(x) \vee \mathit{Turning}(x_1) \Leftarrow \mathit{Connected} \qquad (x \neq x_1). \tag{4}$$

(x, x_1 range over $\{1, 2\}$.)

According to Theorem 1, the description in Figure 1 can be turned into a definite theory with the same set of models by replacing the rule (4) with the following rules [8]:

$$\begin{array}{rcll} \mathit{Turning}(x) & \Leftarrow & \mathit{Turning}(x_1) \wedge \mathit{Connected} & (x \neq x_1) \\ \bot & \Leftarrow & \neg\big(\bigwedge_x \mathit{Turning}(x) \supset \bigvee_x \mathit{MotorOn}(x)\big). & \end{array} \tag{5}$$

For any causal theory T whose rules have the form (1), the completion of $DR(T)$ is equivalent to a set of implications of the following two kinds:

$$G \supset F \tag{6}$$

for every rule (1) in T, and

$$l \ \supset \bigvee_{\substack{F \Leftarrow G \ \in \ T \\ l \in F}} \left(G \wedge \bigwedge_{l' \in F \setminus \{l\}} \overline{l'} \right) \tag{7}$$

for each literal l. Accordingly, the completion of the definite theory

$$DR(T) \cup \left\{ \bot \Leftarrow \neg\Big(F_L \supset \bigvee R(L)\Big) \ : \ L \text{ is a loop of } T \right\}$$

from Corollary 1 is equivalent to the union of sets of formulas (6), (7) and $\{F_L \supset \bigvee R(L) \ : \ L$ *is a loop of* $T\}$.

[8] The other rules obtained from translating (4) can be dropped without changing the set of models.

4 The Relationship between Causal Logic and Logic Programs

We discuss that the semantics of causal logic is closely related to the answer set semantics. This section is not self-contained and readers are assumed to be familiar with [Lee and Lifschitz, 2003b], which defines completion and loop formulas for disjunctive logic programs. In that paper we showed that the answer sets for a disjunctive logic program can be characterized as the models of its completion that satisfy its loop formulas. It is interesting to note that formulas (6), (7) in this paper are very similar to the completion of a disjunctive logic program defined in that paper, and that a loop formula for a causal theory has the same form with a loop formula for a logic program. Comparing Theorem 1 in this note with Theorem 1 from [Lee and Lifschitz, 2003b] tells us that an essential difference between the two formalisms is related to the definition of a loop in each. In the answer set semantics, an edge of the positive dependency graph goes from an atom in the head to a positive atom in the body, while in causal logic an edge of the head dependency graph goes from a literal in the head to the literal complementary to another literal in the head.[9]

4.1 From Logic Programs to Causal Logic

The idea above can be used to prove theorems about the relationship between logic programs and causal logic. For instance, it can be used to get an alternative proof of the lemma from [Giunchiglia *et al.*, 2003, Section 8.3], which shows how to embed disjunctive logic programs into causal logic. Proposition 2 below is slightly more general than the lemma[10] and is a special case of Theorem 1 from [Doğandağ *et al.*, 2003].

We follow the definition of the answer set semantics reviewed in Section 2 of [Lee and Lifschitz, 2003b], which applies to finite programs without classical negation. Fact 3 from [Lee and Lifschitz, 2003b] tells us that any program whose rules may have nested expressions [Lifschitz *et al.*, 1999] both in the heads and the bodies of rules is strongly equivalent to a program whose rules have the form

$$A \leftarrow B, F \tag{8}$$

where A is a disjunction of distinct atoms, B is a conjunction of distinct atoms, and F is a formula in which every occurrence of each atom is in the scope of negation as failure. In the following, we identify an interpretation with the set of atoms that are true in it.

[9] Another difference is that logic programs defined in [Lee and Lifschitz, 2003b] do not allow classical negation but allow negation as failure, while in causal logic it is opposite.

[10] The lemma from [Giunchiglia *et al.*, 2003, Section 8.3] allows infinite causal theories also.

Proposition 2 *Let Π be a (finite) program of a signature σ (without classical negation) whose rules have the form (8). A set X of atoms is an answer set for Π iff X is a model of the causal theory*

$$\{B \supset A \Leftarrow F \ : \ A \leftarrow B, F\} \ \cup \ \{\neg c \Leftarrow \neg c \ : \ c \in \sigma\}.^{11}$$

Proof Let T_Π be the corresponding causal theory for Π. It is easy to check that the completion of Π and the completion of $DR(T_\Pi)$ are equivalent to each other. As for the loop formulas, observe first that every loop L of Π is a loop of T_Π. It is also easy to check that for every loop L of Π, $CLF(L)$ for Π is equivalent to $CLF(L)$ for T_Π. Note that every loop that is in T_Π but not in Π contains at least one negative literal. $CLF(L')$ of these extra loops L' are tautologies due to the presence of $\{\neg c \Leftarrow \neg c \ : \ c \in \sigma\}$: the antecedent of the implication $CLF(L')$ is a conjunction of literals that contains $\neg c \in L'$ and one of the disjunctive terms in the consequent is $\neg c$. By Theorem 1 in this note and by Theorem 1 from [Lee and Lifschitz, 2003b] the proposition follows. ∎

Using the fact that the completion of Π and the completion of $DR(T_\Pi)$ are equivalent to each other, if a logic program Π is tight as defined in [Lee and Lifschitz,2003b], we get a causal theory whose models are exactly the program's answer sets by simply replacing $\leftarrow$ in the logic program with $\Leftarrow$ (assuming the signature of the causal theory contains only atoms used in the logic program) and by adding the closed world assumption for atoms, i.e., $\neg c \Leftarrow \neg c$ for all atoms c.

4.2 Transitive Closure

The difference on the definition of a loop in each formalism can guide us in translating a representation in one formalism to the other. In logic programming the following describes the transitive closure tc of a binary relation p on a set A:

$$\begin{array}{rcl} p(x,y) & & \textit{for any pair } x,y \in A \textit{ such that } p(x,y) \textit{ holds} \\ tc(x,y) & \leftarrow & p(x,y) \\ tc(x,z) & \leftarrow & p(x,y), tc(y,z). \end{array} \tag{9}$$

One might be tempted to write the corresponding representation in causal logic as follows:

$$\begin{array}{rcll} p(x,y) & \Leftarrow & \top & \textit{for any pair } x,y \in A \textit{ such that } p(x,y) \textit{ holds} \\ tc(x,y) & \Leftarrow & p(x,y) & \\ tc(x,z) & \Leftarrow & p(x,y) \wedge tc(y,z) & \\ \neg p(x,y) & \Leftarrow & \neg p(x,y) & \\ \neg tc(x,y) & \Leftarrow & \neg tc(x,y). & \end{array} \tag{10}$$

[11] We agree to identify '*not*' in logic programs with '¬' in causal theories, ',' with '∧', and ';' with '∨'.

Note that the completion of (9) is equivalent to the completion of (10). If p is acyclic, then tc in (10) describes the transitive closure correctly. Otherwise, the representation may allow spurious models that do not correspond to the transitive closure.

The presence of spurious models is related to the cyclic causality in the third rule. The loop formulas for (10) are not equivalent to the loop formulas for (9). In (9) the third rule tells us that the positive dependency graph has edges that go from $tc(x, z)$ to $tc(y, z)$, while in (10) the corresponding rule does not contribute to edges of the head dependency graph. Indeed, there are no loops for (10).

The problem can be corrected by moving $tc(y, z)$ in the third rule from the body to the head, so that the head dependency graph contains the corresponding edges:

$$tc(y, z) \supset tc(x, z) \Leftarrow p(x, y).^{12}$$

The modified causal theory may have more loops than (9), but the loop formulas for these extra loops are tautologies because each of the loops contains at least one negative literal (same reason as in the proof of Proposition 2). Thus it is easy to see that the loop formulas for the modified causal theory are equivalent to the loop formulas for (9). The translation of the causal logic representation of transitive closure to the corresponding logic program provides an alternative proof of Theorem 2 from [Doğandağ *et al.*, 2003], which shows the correctness of the modified casual theory for representing transitive closure. According to Theorem 1, tc can be also described by definite theories using the translation from Section 3.

4.3 Almost Definite Theories

[Doğandağ *et al.*, 2003] defines a class of causal theories called *almost definite*, which is more general than the class of definite theories. The paper describes a translation that turns any almost definite causal theory into a logic program. Theorem 1 from that paper shows the correctness of the translation: an interpretation is a model of an almost definite causal theory iff it is an answer set for the corresponding logic program.

An alternative (easier) proof of that theorem can be given by using Theorem 1 in this note and Theorem 1 from [Lee and Lifschitz, 2003b]: first translate from each of the formalisms to propositional logic and then show that the translations are equivalent to each other.

Acknowledgements. We are grateful to Selim Erdoğan, Paolo Ferraris, Vladimir Lifschitz and Hudson Turner for useful discussions related to the subject of this paper. This work was partially supported by the Texas Higher Education Coordinating Board under Grant 003658-0322-2001.

[12] According to Proposition 2, we can also write $p(x, y) \wedge tc(y, z) \supset tc(x, z) \Leftarrow \top$. Since $p(x, y)$ does not contribute to any loops, moving $p(x, y)$ from the head to the body does not change loop formulas. The case is similar with the second rule of (10).

References

Akman *et al.*, 2003. Varol Akman, Selim Erdoğan, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Representing the Zoo World and the Traffic World in the language of the Causal Calculator.[13] *Artificial Intelligence*, 2003. To appear.

Artikis *et al.*, 2003a. A. Artikis, M. Sergot, and J. Pitt. An executable specification of an argumentation protocol. In *Proceedings of Conference on Artificial Intelligence and Law (ICAIL)*, pages 1–11. ACM Press, 2003.

Artikis *et al.*, 2003b. A. Artikis, M. Sergot, and J. Pitt. Specifying electronic societies with the Causal Calculator. In F. Giunchiglia, J. Odell, and G. Weiss, editors, *Proceedings of Workshop on Agent-Oriented Software Engineering III (AOSE)*, LNCS 2585. Springer, 2003.

Campbell and Lifschitz, 2003. Jonathan Campbell and Vladimir Lifschitz. Reinforcing a claim in commonsense reasoning.[14] In *Working Notes of the AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 2003.

Chopra and Singh, 2003. Amit Chopra and Munindar Singh. Nonmonotonic commitment machines. In *Agent Communication Languages and Conversation Policies AAMAS 2003 Workshop*, 2003.

Clark, 1978. Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

Doğandağ *et al.*, 2003. Semra Doğandağ, Paolo Ferraris, and Vladimir Lifschitz. Almost definite causal theories. In this volume.

Giunchiglia *et al.*, 2003. Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories.[15] *Artificial Intelligence*, 2003. To appear.

Lee and Lifschitz, 2003a. Joohyung Lee and Vladimir Lifschitz. Describing additive fluents in action language $\mathcal{C}+$[16] In *Proc. IJCAI-03*, pages 1079–1084, 2003.

Lee and Lifschitz, 2003b. Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs[17] In *Proc. ICLP-03*, 2003. To appear.

Lifschitz *et al.*, 1999. Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

Lifschitz *et al.*, 2000. Vladimir Lifschitz, Norman McCain, Emilio Remolina, and Armando Tacchella. Getting to the airport: The oldest planning problem in AI. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 147–165. Kluwer, 2000.

Lifschitz, 2000. Vladimir Lifschitz. Missionaries and cannibals in the Causal Calculator. In *Principles of Knowledge Representation and Reasoning: Proc. Seventh Int'l Conf.*, pages 85–96, 2000.

Lin and Zhao, 2002. Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAAI-02*, 2002.

McCain and Turner, 1997. Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.

McCain, 1997. Norman McCain. *Causality in Commonsense Reasoning about Actions.*[18] PhD thesis, University of Texas at Austin, 1997.

[13] `http://www.cs.utexas.edu/users/appsmurf/papers/zt.ps.`

[14] `http://www.cs.utexas.edu/users/vl/papers/sams.ps .`

[15] `http://www.cs.utexas.edu/users/vl/papers/nmct.ps.`

[16] `http://www.cs.utexas.edu/users/appsmurf/papers/additive-ijcai.ps.`

[17] `http://www.cs.utexas.edu/users/appsmurf/papers/disjunctive.ps.`

[18] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.Z.`

Logic Programs With Monotone Cardinality Atoms

Victor W. Marek[1], Ilkka Niemelä[2], and Mirosław Truszczyński[1]

[1] Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA
[2] Department of Computer Science and Engineering
Helsinki University of Technology,
P.O.Box 5400, FIN-02015 HUT, Finland

Abstract. We investigate *mca-programs*, that is, logic programs with clauses built of monotone cardinality atoms of the form kX, where k is a non-negative integer and X is a finite set of propositional atoms. We develop a theory of mca-programs. We demonstrate that the operational concept of the one-step provability operator generalizes to mca-programs, but the generalization involves nondeterminism. Our main results show that the formalism of mca-programs is a common generalization of (1) normal logic programming with its semantics of models, supported models and stable models, (2) logic programming with cardinality atoms and with the semantics of stable models, as defined by Niemelä, Simons and Soininen, and (3) of disjunctive logic programming with the *possible-model* semantics of Sakama and Inoue.

1 Introduction

We introduce and study logic programs whose clauses are built of *monotone cardinality atoms* (*mc-atoms*), that is, expressions of the form kX, where k is a non-negative integer and X is a finite set of propositional atoms. Intuitively, kX is true in an interpretation M if at least k atoms in X are true in M. Thus, the intended role for mc-atoms is to represent constraints on lower bounds of cardinalities of sets. We refer to programs with mc-atoms as *mca-programs*. We are motivated in this work by the recent emergence and demonstrated effectiveness of logic programming extended with means to model cardinality constraints [12, 11,15], and by the need to develop sound theoretical basis for such formalisms.

In the paper, we develop a theory of mca-programs. In that we closely follow the development of normal logic programming and lift all its major concepts, techniques and results to the setting of mca-programs. There is, however, a basic difference. Mc-atoms have, by their very nature, a built-in nondeterminism. They can be viewed as shorthands for certain disjunctions and, in general, there are many ways to make an mc-atom kX true. This nondeterminism has a key consequence. The one-step provability operator is no longer deterministic, as in normal logic programming, where it maps interpretations to interpretations. In

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 154–166, 2004.

the case of mca-programs, the one-step provability operator is nondeterministic. It assigns to an interpretation M a *set* of interpretations, each regarded as possible and equally likely outcomes of applying the operator to M.

Modulo this difference, our theory of mca-programs parallels that of normal logic programs. First, we introduce *models* and *supported* models of an mca-program and describe them in terms of the one-step provability operator in much the same way it is done in normal logic programming. To define *stable* models we first define the class of *Horn* mca-programs by disallowing the negation operator in the bodies of clauses. We show that the nondeterministic one-step provability operator associates with Horn mca-programs a notion of a (nondeterministic) computation (the counterpart to the bottom-up computation with normal Horn programs) and a class of *derivable models* (counterparts to the least model of a normal Horn program). We then lift the notion of the Gelfond-Lifschitz reduct [8] to the case of mca-programs and define a stable model of an mca-program as a set of atoms that is a derivable model of the reduct. A striking aspect of our construction is that all its steps are *literal* extensions of the corresponding steps in the original approach. We show that stable models behave as expected. They are supported and, in case of Horn mca-programs, derivable.

An intended meaning of an mc-atom $1\{a\}$ is that a be true. More formally, $1\{a\}$ is true in an interpretation if and only if a is true in that interpretation. That connection implies a natural representation of normal logic programs as mca-programs. We show that this representation preserves all semantics we discuss in the paper. It follows that the formalism of mca-programs can be viewed as a direct generalization of normal logic programming.

As we noted, an extension of logic programming with direct ways to model cardinality constraints was first proposed in [12]. That work defined a syntax of logic programs with cardinality constraints (in fact, with more general *weight constraints*) and introduced the notion of a *stable model*. We will refer to programs in that formalism as *NSS-programs*. One of the results in [12] showed that NSS-programs generalized normal logic programming with the stable-model semantics of Gelfond and Lifschitz [8]. However, the notion of the reduct underlying the definition of a stable model given in [12] is different from that proposed by Gelfond and Lifschitz [8] and the precise nature of the relationship between normal logic programs and NSS-programs was not clear.

Mca-programs explicate this relationship. We show that the formalism of mca-programs parallels normal logic programming. In particular, major concepts, results and techniques in normal logic programming have counterparts in the setting of mca-programs. We also prove that under some simple transformations, NSS-programs are equivalent to mca-programs. Through this connection, the theory of normal logic programming can be lifted to the setting of NSS-programs leading to new characterizations of stable models of NSS-programs.

Finally, we show that mca-programs not only provide an overarching framework for both normal logic programs and NSS-programs. They are also useful in investigating disjunctive logic programs. In the paper, we show that logic

programming with mc-atoms generalize disjunctive logic programming with the possible-model semantics introduced in [14].

2 Logic Programs With Monotone Cardinality Atoms

Let At be a set of (propositional) *atoms*. An *mc-atom over* At (short for a *monotone cardinality atom over* At) is any expression of the form kX, where k is a non-negative integer and $X \subseteq At$ is a *finite* set such that $k \leq |X|$. We call X the *atom set* of an mc-atom $A = kX$ and denote it by $aset(A)$. An intuitive reading of an mc-atom kX is: *at least* k *atoms in* X *are true*. The intended meaning of kX explains the requirement that $k \leq |X|$. Clearly, if $k > |X|$, it is impossible to have in X at least k true atoms and the expression kX is equivalent to a contradiction.

An *mc-literal* is an expression of the form A or $\mathbf{not}(A)$, where A is an mc-atom. An *mca-clause* (short for a *monotone-cardinality-atom clause*) is an expression r of the form

$$H \leftarrow L_1, \ldots, L_m, \tag{1}$$

where H is an mc-atom and L_i, $1 \leq i \leq m$, are mc-literals. We call the mc-atom H the *head* of r and denote it by $hd(r)$. We call the set $\{L_1, \ldots, L_m\}$ the *body* of r and denote it by $bd(r)$. An mca-clause is *Horn* if its body does not contain literals of the form $\mathbf{not}(A)$. Finally, for an mca-clause r, we define the *head set* of r, $hset(r)$, by setting $hset(r) = aset(hd(r))$.

Mca-clauses form *mca-programs*. We define the *head set* of an mca-program P, $hset(P)$, by $hset(P) = \bigcup\{hset(r)\colon r \in P\}$ (if $P = \emptyset$, $hset(P) = \emptyset$, as well). If all clauses in an mca-program P are Horn, P is a *Horn* mca-program.

One can give a declarative interpretation to mca-programs in terms of a natural extension of the semantics of propositional logic. We say that a set M of atoms *satisfies* an mc-atom kX if $|M \cap X| \geq k$, and M *satisfies* an mc-literal $\mathbf{not}(kX)$ if it does not satisfy kX (that is, if $|M \cap X| < k$). A set of atoms M satisfies an mca-clause (1) if M satisfies H whenever M satisfies all literals L_i, $1 \leq i \leq m$. Finally, a set of atoms M satisfies an mca-program P if it satisfies all clauses in P. We often say "is a model of" instead of "satisfies". We use the symbol $\models$ to denote the satisfaction relation.

The following straightforward property of mc-atoms explains the use of the term "monotone" in their name.

Proposition 1. *Let A be an mc-atom over a set of atoms At. For every sets $M, M' \subseteq At$, if $M \subseteq M'$ and $M \models A$ then $M' \models A$.*

Mca-clauses also have a procedural interpretation in which they are viewed as derivation rules. Intuitively, if an mca-clause r has its body satisfied by some set of atoms M, then r provides *support* for deriving from M any set of atoms M' such that

1. M' consists of atoms mentioned in the head of r (r provides no grounds for deriving atoms that do not appear in its head)

2. M' satisfies the head of r (since r "fires", the constraint imposed by its head must hold).

Clearly, the process of deriving M' from M by means of r is *nondeterministic* in the sense that, in general, there are several sets that are supported by r and M.

This notion of nondeterministic derivability extends to programs and leads to the concept of the nondeterministic one-step provability operator. Let P be an mca-program and let $M \subseteq At$ be a set of atoms. We set $P(M) = \{r \in P\colon M \models bd(r)\}$. We call mca-clauses in $P(M)$, *M-applicable.*

Definition 1. *Let P be an mca-program and let $M \subseteq At$. A set M' is* nondeterministically one-step provable *from M by means of P, if $M' \subseteq hset(P(M))$ and $M' \models hd(r)$, for every mca-clause r in $P(M)$.*

The nondeterministic one-step provability operator *T_P^{nd}, is a function from $\mathcal{P}(At)$ to $\mathcal{P}(\mathcal{P}(At))$ and such that for every $M \subseteq At$, $T_P^{nd}(M)$ consists all sets M' that are nondeterministically one-step provable from M by means of P.*

As we indicate next, for every $M \subseteq At$, $T_P^{nd}(M)$ is nonempty. It follows that T_P^{nd} can be viewed as a formal representation of a *nondeterministic* operator on $\mathcal{P}(At)$, which assigns to every subset M of At a subset of At arbitrarily selected from the collection $T_P^{nd}(M)$ of possible outcomes. Since $T_P^{nd}(M)$ is nonempty, this nondeterministic operator is well defined.

Proposition 2. *Let P be an mca-program and let $M \subseteq At$. Then, $hset(P(M)) \in T_P^{nd}(M)$. In particular, $T_P^{nd}(M) \neq \emptyset$.*

The operator T_P^{nd} plays a fundamental role in our research. It allows us to formalize procedural interpretations of mca-clauses and identify for them matching classes of models that provide the corresponding declarative account.

Our first result characterizes models of mca-programs. This characterization is a generalization of the familiar description of models of normal logic programs as prefixpoints of T_P.

Theorem 1. *Let P be an mca-program and let $M \subseteq At$. The set M is a model of P if and only if there is $M' \in T_P^{nd}(M)$ such that $M' \subseteq M$.*

A straightforward corollary states that every mca-program has a model.

Corollary 1. *Let P be an mca-program. Then, $hset(P)$ is a model of P.*

Models of mca-programs may contain elements that have no support in a program and the model itself. For instance, let us consider an mca-program P consisting of the clause: $1\{p, q\} \leftarrow \mathbf{not}(1\{q\})$, where p and q are two different atoms. Let $M_1 = \{q\}$. Clearly, M_1 is a model of P. However, M_1 has no support in P and itself. Indeed, $T_P^{nd}(M_1) = \{\emptyset\}$ and so, P and M_1 do not provide support for any atom. Similarly, another model of P, the set $M_2 = \{p, r\}$, where $r \in At$ is an atom different from p and q, has no support in P and itself. We have $T_P^{nd}(M_2) = \{\{p\}, \{q\}, \{p, q\}\}$ and so, p has support in P and M_2, but r does not. Finally, the set $M_3 = \{p\}$, which is also a model of P, *has support* in P and itself. Indeed, $T_P^{nd}(M_3) = \{\{p\}, \{q\}, \{p, q\}\}$ and there is a way to derive M_3 from P and M_3. We formalize now this discussion in the following definition.

Definition 2. *Let P be an mca-program. A set of atoms M is a* supported model *of P if $M \in T_P^{nd}(M)$.*

The use of the term "model" is justified. By Theorem 1, supported models of P are indeed models of P, as stated in the following result.

Corollary 2. *Every supported model of an mca-program P is a model of P.*

Finally, we have the following characterization of supported models.

Proposition 3. *Let P be an mca-program. A set $M \subseteq At$ is a supported model of P if and only if M is a model of P and $M \subseteq hset(P(M))$.*

3 Horn mca-Programs

To introduce *stable* models of mca-programs, we need first to study Horn mca-programs. With each Horn mca-program P one can associate the concept of a *P-computation*. Namely, a *P-computation* is a sequence $(X_n)_{n=0,1,\ldots}$ such that $X_0 = \emptyset$ and, for every non-negative integer n,

1. $X_n \subseteq X_{n+1}$, and
2. $X_{n+1} \in T_P^{nd}(X_n)$.

Given a computation $t = (X_n)_{n=0,1,\ldots}$, we call $\bigcup_{n=0}^{\infty} X_n$ the *result* of the computation t and denote it by R_t.

Proposition 4. *Let P be a Horn mca-program and let t be a P-computation. Then $R_t \subseteq hset(P(R_t))$.*

If P is a Horn mca-program then P-computations exist. Let M be a model of P. We define the sequence $t^{P,M} = (X_n^{P,M})_{n=0,1,\ldots}$ as follows. We set $X_0^{P,M} = \emptyset$ and, for every $n \geq 0$, $X_{n+1}^{P,M} = hset(P(X_n^{P,M})) \cap M$.

Theorem 2. *Let P be a Horn mca-program and let $M \subseteq At$ be its model. The sequence $t^{P,M}$ is a P-computation.*

We call the P-computation $t^{P,M}$ the *canonical* P-computation for M. Since every mca-program P has models, we obtain the following corollary.

Corollary 3. *Every Horn mca-program has at least one computation.*

The results of computations are supported models (and, thus, also models) of Horn mca-programs.

Proposition 5. *Let P be a Horn mca-program and let t be a P-computation. Then, the result of t, R_t, is a supported model of P.*

We use the concept of a computation to identify a certain class of models of Horn mca-programs.

Definition 3. *Let P be a Horn mca-program. We say that a set of atoms M is a* derivable model *of P if there exists a P-computation t such that $M = R_t$.*

Derivable models can be obtained as results of their own canonical computations.

Proposition 6. *Let M be a derivable model of a Horn mca-program P. Then $M = R_{t^{P,M}}$.*

Proposition 5 and Theorem 2 entail several properties of Horn mca-programs, their computations and models. We gather them in the following corollary.

Corollary 4. *Let P be a Horn mca-program. Then:*

1. *P has at least one derivable model.*
2. *P has a largest derivable model.*
3. *Every derivable model of P is a supported model of P.*
4. *For every model M of P there is a derivable model M' of P such that $M' \subseteq M$.*
5. *Every minimal model of P is derivable.*

4 Stable Models of mca-Programs

We will now use the results of the two previous sections to introduce and study the class of *stable* models of mca-programs.

Definition 4. *Let P be an mca-program and let $M \subseteq At$. The* reduct *of P with respect to M, P^M in symbols, is a Horn mca-program obtained from P by (1) removing from P every clause containing in the body a literal $\mathbf{not}(A)$ such that $M \models A$, and (2) removing all literals of the form $\mathbf{not}(A)$ from all remaining clauses in P. A set of atoms M is a* stable *model of P if M is a derivable model of the reduct P^M.*

Stable models of an mca-program P are indeed models of P. Thus, the use of the term "model" in their name is justified. In fact, a stronger property holds: stable models of mca-programs are supported.

Proposition 7. *Let P be an mca-program. If $M \subseteq At$ is a stable model of P then M is a supported model of P.*

With the notion of a stable model in hand, we can strengthen Proposition 5.

Proposition 8. *Let P be a Horn mca-program. A set of atoms $M \subseteq At$ is a derivable model of P if and only if M is a stable model of P.*

We will now describe a procedural characterization of stable models of mca-programs, relying on a notion of a computation related to but different from the one we discussed in Section 3 in the context of Horn programs. A difference is that now at each stage in a computation we must make sure that once a clause is applied, it remains applicable at *any* stage of the process. It is not *a priori* guaranteed due to the presence of negation in the bodies of general mca-clauses.

A formal definition is as follows. Let P be an mca-program. A sequence $\varepsilon = (X_n)_{n=0,1,2,\ldots}$ is a *quasi P-computation*, if $X_0 = \emptyset$ and if for every $n = 0, 1, \ldots$ there is a clause $r_n \in P$ such that

1. $X_n \models bd(r_n)$.
2. there is $X \subseteq hset(r_n)$ such that $X \models hd(r_n)$ and $X_{n+1} = X_n \cup X$ (this X is what is "computed" by applying r_n).
3. for every $i = 0, 1 \ldots, n$ and for every mc-atom kX occurring negated in $bd(r_i)$, $X_{n+1} \not\models kX$.

We call the set $\bigcup_{1 \leq k < \omega} X_k$ the *result* of the quasi P-computation ε.

Theorem 3. *A set of atoms M is a stable model of P if and only if M is a model of P and for some quasi P-computation ε, M is the result of ε.*

Theorem 3 states that if we apply clauses *carefully*, making sure that at no stage we satisfy an mc-atom appearing negated in clauses applied so far (including the one selected to apply at the present stage) and we ever compute a model in this way, then this model is a stable model of P. Conversely, every stable model can be obtained as a result of such a *careful* computation.

5 Extension of mca-Programs by Constraint mca-Clauses

We can extend the language of mca-programs by allowing clauses with the empty head. Namely, we define a *constraint mca-clause* to be an expression r of the form

$$\leftarrow L_1, \ldots, L_m, \tag{2}$$

where L_i, $1 \leq i \leq m$, are mc-literals.

The notion of satisfiability that we introduced for mca-clauses extends to the case of mca-constraints. A set of atoms M *satisfies* a constraint r if there is a literal $L \in bd(r)$ such that $M \not\models L$. We can now extend the definitions of supported and stable models to the more general class of mca-programs with constraint mca-clauses as follows.

Definition 5. *Let P be an mca-program with constraint mca-clauses. A set of atoms M is a* supported *(*stable*) model of P if M is a supported (stable) model of P', where P' consists of all non-constraint mca-clauses in P, and if M is a model of all constraint mca-clauses in P.*

Let us observe that several of our earlier results such as Proposition 7 and Theorem 3 lift *verbatim* to the case of programs with constraints.

6 Mca-Programs and Normal Logic Programming

An mc-atom $1\{a\}$ is true in a model M if and only if a is true in M. Thus, intuitively, $1\{a\}$ and a are equivalent. That suggests a way to interpret normal clauses and programs as mca-clauses and mca-programs. Let

$$r = \quad c \leftarrow a_1, \ldots, a_m, \mathbf{not}(b_1), \ldots, \mathbf{not}(b_n).$$

By $mca(r)$ we mean the mc-clause

$$1\{c\} \leftarrow 1\{a_1\}, \ldots, 1\{a_m\}, \mathbf{not}(1\{b_1\}), \ldots, \mathbf{not}(1\{b_n\}).$$

(If all a_i and all b_i are distinct, which we can assume without loss of generality, a simpler translation, $1\{c\} \leftarrow m\{a_1, \ldots, a_m\}, \mathbf{not}(1\{b_1, \ldots, b_n\})$, could be used.) Moreover, given a normal program P, we set $mca(P) = \{mc(r) \colon r \in P\}$.

This encoding interprets normal logic programs as mca-programs so that basic properties and concepts of normal logic programming can be viewed as special cases of properties and concepts in mca-programming. In the following theorem, we gather several results establishing appropriate correspondences.

Theorem 4. *Let P be a normal logic program and let M be a set of atoms.*

1. *P is a Horn program if and only if $mca(P)$ is a Horn mca-program.*
2. *If P is a Horn program then the least model of P is the only derivable model of $mca(P)$.*
3. *$\{T_P(M)\} = T^{nd}_{mca(P)}(M)$.*
4. *$mca(P^M) = mca(P)^M$.*
5. *M is a model (supported model, stable model) of P if and only if M is a model (supported model, stable model) of $mca(P)$.*

Finally, we identify a class of mca-programs, which offers a most direct generalization of normal logic programming.

Definition 6. *An mca-clause r is* deterministic *if $hd(r) = 1\{a\}$, for some atom a. An mca-program is* deterministic *if every clause in P is deterministic.*

The intuition behind the term is clear. If the head of an mca-clause is of the form $1\{a\}$, then there is only one possible effect of applying the clause: a has to be concluded. Thus, the nondeterminism that arises in the context of arbitrary mc-atoms disappears. Formally, we capture this property in the following result.

Proposition 9. *Let P be a deterministic mca-program. Then, for every set of atoms M, $T^{nd}_P(M) = \{M'\}$, for some set of atoms M'.*

Thus, for a deterministic mca-program P, the operator T^{nd}_P is deterministic and, so, can be regarded as an operator with both the domain and codomain $\mathcal{P}(At)$. We will write T^d_P, to denote it. Models, supported models and stable models of a deterministic mca-program can be introduced in terms of the operator T^d_P in exactly the same way the corresponding concepts are defined in normal

logic programming. In particular, the algebraic treatment of logic programming developed in [7,13,2] applies literally to deterministic mca-programs and results in a natural and direct extension of normal logic programming. We will explicitly mention just one result here that will be of importance later in the paper.

Proposition 10. *Let P be a deterministic Horn program. Then P has exactly one derivable model and this model is the least model of P.*

7 Mca-Programs and NSS-Programs

We will first briefly review the concept of an NSS-program [12], the semantics of stable models of such programs, as introduced in [12], and then relate this formalism to that of mca-programs.

A *cardinality atom* (c-atom, for short) is an expression of the form kXl, where $X \subseteq At$, and l and k are integers such that $0 \leq k \leq l \leq |X|$. We call X an *atom set* of a c-atom $A = kXl$ and, as before, we denote it by $aset(A)$[1].

We say that a set of atoms M satisfies a c-atom kXl if $k \leq |M \cap X| \leq l$ ($M \models kXl$, in symbols). It is clear that when $k = 0$ or $l = |X|$, the corresponding inequality is trivially true. Thus, we omit from the notation k, if equal to 0, and l, if equal to $|X|$.

A *cardinality-atom clause* (ca-clause, for short) is an expression r of the form

$$A \leftarrow B_1, \ldots, B_n,$$

where A and B_i, $1 \leq i \leq n$, are c-atoms. We call A the head of r and $\{B_1, \ldots, B_n\}$ the *body* of r. We denote them by $hd(r)$ and $bd(r)$, respectively. A *ca-program* is a collection of ca-clauses.

We say that a set $M \subseteq At$ *satisfies* a ca-clause r if M satisfies $hd(r)$ whenever it satisfies each c-atom in the body of r. We say that M satisfies a ca-program P if M satisfies each ca-clause in P. We write $M \models r$ and $M \models P$ in these cases, respectively.

We will now recall the concept of a stable model of a ca-program [12]. Let P be an NSS-program and let $M \subseteq At$. By the *NSS-reduct* of P with respect to M we mean the NSS-program obtained by:

1. eliminating from P every clause r such that $M \not\models B$, for at least one c-atom $B \in bd(r)$.
2. replacing each remaining ca-clause $r = kXl \leftarrow k_1Y_1l_1, \ldots k_nY_nl_n$ with all clauses of the form $1\{a\} \leftarrow k_1Y_1, \ldots, k_nY_n$, where $a \in X \cap M$.

With some abuse of notation, we denote the resulting program by P^M (the type of the program determines which reduct we have in mind). It is clear that P^M is a deterministic Horn mca-program. Thus, it has a least model, $lm(P^M)$.

[1] To be precise, [12] allows also for negated atoms to appear as elements of X. One can eliminate occurrences of negative literals by introducing new atoms. Thus, for this work, we decided to restrict the syntax of NSS-programs.

Definition 7. *Let P be a ca-program. A set $M \subseteq At$ is a* stable model *of P if $M = lm(P^M)$ and $M \models P$.*

We will now show that the formalisms of mca-programs and ca-programs with their corresponding stable-model semantics are equivalent. We start by describing an encoding of ca-clauses and ca-programs by mca-clauses and mca-programs. To simplify the description of the encoding and make it uniform, we assume that all bounds are present (we recall that whenever any of the bounds are missing from the notation, they can be introduced back). Let r be the following ca-clause: $kXl \leftarrow k_1X_1l_1, \ldots, k_mX_ml_m$. We represent this ca-clause by a pair of mca-clauses, $e^1_{mca}(r)$ and $e^2_{mca}(r)$ that we define as the following two mca-clauses, respectively:

$$kX \leftarrow k_1X_1, \ldots, k_mX_m, \mathbf{not}((l_1+1)X_1), \ldots, \mathbf{not}((l_m+1)X_m),$$

and

$$\leftarrow (l+1)X, k_1X_1, \ldots, k_mX_m, \mathbf{not}((l_1+1)X_1), \ldots, \mathbf{not}((l_m+1)X_m).$$

Given a ca-program P, we translate it into an mca-program

$$e_{mca}(P) = \bigcup_{r \in P} \{e^1_{mca}(r), e^2_{mca}(r)\}.$$

Theorem 5. *Let P be a ca-program. A set of atoms M is a stable model of P, as defined for ca-programs, if and only if M is a stable model of $e_{mca}(P)$, as defined for mca-programs.*

This theorem shows that the formalism of mca-programs is at least as expressive as that of ca-programs. The converse is true as well: ca-programs are at least as expressive as mca-programs. Let r be the following mca-clause:

$$kX \leftarrow k_1X_1, \ldots, k_mX_m, \mathbf{not}(l_1Y_1), \ldots, \mathbf{not}(l_nX_n).$$

We define $e_{ca}(r)$ as follows. If there is i, $1 \leq i \leq n$, such that $l_i = 0$, we set $e_{ca}(r) = \quad kX \leftarrow kX$ (in fact any tautology would do). Otherwise, we set

$$e_{ca}(r) = \quad kX \leftarrow k_1X_1, \ldots, k_mX_m, Y_1(l_1-1), \ldots, Y_n(l_n-1).$$

Given an mca-program P, we define $e_{ca}(P) = \{e_{ca}(r) \colon r \in P\}$.

Theorem 6. *Let P be an mca-program. A set of atoms M is a stable model of P, as defined for mca-programs, if and only if M is a stable model of $e_{ca}(P)$, as defined for ca-programs.*

Theorems 5 and 6 establish the equivalence of ca-programs and mca-programs with respect to the stable model semantics. The same translations also preserve the concept of a model. Finally, Theorem 5 suggests a way to introduce the notion of a supported model for a ca-program: a set of atoms M is defined to

be a *supported* model of a ca-program P if it is a supported model of the mca-program $e_{mca}(P)$. With this definition, the two translations e_{mca} and e_{ca} also preserve the concept of a supported model.

We also note that this equivalence demonstrates that ca-programs with the semantics of stable models as defined in [12] can be viewed as a generalization of normal logic programming. It follows from Theorems 4 and 6 that the encoding of normal logic programs as ca-programs, defined as the composition of the translations *mca* and e_{ca}, preserves the semantics of models, supported models and stable models (an alternative proof of this fact, restricted to the case of stable models only was first given in [12] and served as a motivation for the class of ca-programs and its stable-model semantics). This result is important, as it is not at all evident that the NSS-reduct and Definition 7 generalize the semantics of stable models as defined in [8].

Given that the formalisms of ca-atoms and mca-atoms are equivalent, it is important to stress what differs them. The advantage of the formalism of ca-programs is that it does not require the negation operator in the language. The strength of the formalism of mca-programs lies in the fact that its syntax so closely resembles that of normal logic programs, and that the development of the theory of mca-programs so closely follows that of the normal logic programming.

8 Mca-Programs and Disjunctive Logic Programs

The formalism of mca-programs also extends an approach to disjunctive logic programming, proposed in [14]. In that paper, the authors introduced and investigated a semantics of *possible models* for disjunctive logic programs. We will now show that disjunctive programming with the semantics of possible models is a special case of the logic mca-programs with the semantics of stable models.

Let r be a disjunctive logic program clause of the form:

$$c_1 \vee \ldots \vee c_k \leftarrow a_1, \ldots, a_m, \mathbf{not}(b_1), \ldots, \mathbf{not}(b_n),$$

where all a_i, b_i and c_i are atoms. We define an mca-clause

$$mca_d(r) = \quad 1\{c_1, \ldots, c_k\} \leftarrow 1\{a_1\}, \ldots, 1\{a_m\}, \mathbf{not}(1\{b_1\}), \ldots, \mathbf{not}(1\{b_n\}).$$

For a disjunctive logic program P, we define $mca_d(P) = \{mca_d(r) \colon r \in P\}$. We have the following theorem.

Theorem 7. *Let P be a disjunctive logic program. A set of atoms M is a possible model of P if and only if M is a stable model of the mca-program $mca_d(P)$.*

We also note that there are strong analogies between the approach we propose here and some of the techniques discussed in [14]. In particular, [14] presents a computational procedure for disjunctive programs without negation that is equivalent to our notion of a P-computation. We stress however, that the class of mca-programs is more general and that our approach, consistently exploiting properties of an operator T_P^{nd}, is better aligned with a standard development of normal logic programming.

9 Discussion

Results of our paper point to a central position of mca-programs among other logic programming formalisms. First, mca-programs form a natural generalization of normal logic programs, with most concepts and techniques closely patterned after their counterparts in normal logic programming. Second, mca-programs with the stable-model semantics generalize disjunctive logic programming with the possible-model semantics of [14]. Third, mca-programs provide direct means to model cardinality constraints, a feature that has become broadly recognized as essential to computational knowledge representation formalisms. Moreover, it turns out that mca-programs are, in a certain sense that we made precise in the paper, equivalent, to logic programs with cardinality atoms proposed and studied in [12]. Thus, mca-programs provide a natural link between normal logic programs and the formalism of [12], and help explain the nature of this relationship, hidden by the original definitions in [12].

In this paper, we outlined only the rudiments of the theory of mca-programs. There are several questions that follow from our work and that deserve more attention. First, our theory can be extended to the case of programs built of *monotone-weight atoms*, that is, expressions of the form $a\{p_1 : w_1, \ldots, p_k : w_k\}$, where $a, w_1, \ldots w_k$ are non-negative reals and $p_1, \ldots, p_k$ are propositional atoms. Intuitively, such an atom is satisfied by an interpretation (set of atoms) M if the sum of weights assigned to atoms in $M \cap \{p_1, \ldots, p_k\}$ is at least a.

Next, there is a question whether Fages lemma [6] generalizes to mca-programs. If so, for some classes of programs, one could reduce stable-model computation to satisfiability checking for propositional theories with cardinality atoms [4,9]. That, in turn, might lead to effective computational methods, alternative to direct algorithms such as *smodels* [10] and similar in spirit to the approach of *cmodels* [5,1].

Another interesting aspect concerns some syntactic modifications and "normal form representations" for mca-programs. For instance, at a cost of introducing new atoms, one can rewrite any mca-program into a *simple* mca-program in which every mca-clause contains at most one mca-literal in its body and in which the use of negation is restricted (but not eliminated). We will present these results in a full version of the paper.

The emergence of a nondeterministic one-step provability operator is particularly intriguing. It suggests that, as in the case of normal logic programming [7,13], the theory of mca-programs can be developed by algebraic means. For that to happen, one would need techniques for handling nondeterministic operators on lattices, similar to those presented in the deterministic operators in [2, 3]. That approach might ultimately lead to a generalization of the well-founded semantics to the case of mca-programs.

Acknowledgments. The second author was supported by the Academy of Finland grant 53695. The other two authors were supported by the NSF grants IIS-0097278 and IIS-0325063.

References

1. Y. Babovich and V. Lifschitz. *Cmodels*, 2002. `http://www.cs.utexas.edu/users/tag/cmodels.html`.
2. M. Denecker, V. Marek, and M. Truszczyński. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 127–144. Kluwer Academic Publishers, 2000.
3. M. Denecker, V. Marek, and M. Truszczyński. Ultimate approximations in nonmonotonic knowledge representation systems. In *Principles of Knowledge Representation and Reasoning, Proceedings of the Eighth International Conference (KR2002)*, pages 177–188. Morgan Kaufmann Publishers, 2002.
4. D. East and M. Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, volume 2174, pages 138–153. Lecture Notes in Artificial Intelligence, Springer Verlag, 2001.
5. E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.
6. F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
7. M. C. Fitting. Fixpoint semantics for logic programming – a survey. *Theoretical Computer Science*, 278:25–51, 2002.
8. 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.
9. L. Liu and M. Truszczyński. Local-search techniques in propositional logic extended with cardinality atoms. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming, CP-2003*. Lecture Notes in Computer Science, Springer Verlag, 2003.
10. I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of JICSLP-96*. MIT Press, 1996.
11. 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.
12. I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proceedings of LPNMR-1999*, volume 1730 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1999.
13. T.C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–464, 1990.
14. C. Sakama and K. Inoue. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning*, 13:145–172, 1984.
15. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

Set Constraints in Logic Programming

Victor W. Marek[1] and Jeffrey B. Remmel[2]

[1] Department of Computer Science
University of Kentucky
Lexington, KY 40506, USA
marek@cs.uky.edu
[2] Department of Mathematics
University of California
La Jolla, CA 92093, USA
jremmel@ucsd.edu

Abstract. We investigate a generalization of weight-constraint programs with stable semantics, as implemented in the ASP solver *smodels*. Our programs admit atoms of the form $\langle X, \mathcal{F}\rangle$ where X is a finite set of propositional atoms and $\mathcal{F}$ is an arbitrary family of subsets of X. We call such atoms *set constaints* and show that the concept of stable model can be generalized to programs admitting set constraints both in the bodies and the heads of clauses. Natural tools to investigate the fixpoint semantics for such programs are *nondeterministic* operators in complete lattices. We prove two fixpoint theorems for such operators.

1 Introduction

This paper is concerned with extensions of the Answer Set Programming (ASP) paradigm [SK92,CMT96,NS97,ELM$^+$98,KS99,MT99,NS00,ASP01,Ba03]. These extensions allow a programmer to use various type of aggregation expressions in both the head and the body of clauses of a program. Under the ASP paradigm, a problem is encoded as a program in a declarative language so that the preferred models of the program encode the solutions to the problem. A typical example of an ASP formalism is DATALOG$^\neg$ where the set of preferred models is the set of stable models of the program. In this case, one can use a solver such as *smodels* [NS00] to compute the preferred answers. In [MR02], we studied an extension of DATALOG$^\neg$ called cardinality constraint programming developed by Niemelä, Simons, and Soininen in [NSS99,NS00]. In cardinality constraint (CC) programming one allows atoms of the form kXl, where $k \leq l$ are non-negative integers and X is a finite set of atoms, to appear in both the head and the body of clauses. The meaning of the atom kXl is "at least k but not more than l of atoms from X belong to the intended model M". In fact, the work of [NSS99, NS00] allows for more general weight constraint atoms. That is, if $wt(\cdot)$ is a nonnegative rational valued function on the underlying set of literals of a logic program, then we interpret kXl to mean that in an intended model M of P, $k \leq [\sum_{a\in X\cap M} wt(a)+\sum_{a\in X\setminus M)} wt(\neg a)] \leq l$. These extensions have been implemented in *smodels*, see [NS00]. The main purpose of this paper is to introduce

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 167–179, 2004.

an extension of DATALOG$^\neg$ which we call *set constraint programming* which incorporates both cardinality constraint atoms and weight constraint atoms as a special case.

In [NSS99], Niemelä, Simons, and Soininen defined a natural analogue of stable models for CC-logic programs which we called CC-stable models in [MR02]. The construction of [NSS99] significantly generalizes an older proposal due to Sakama and Inoue [SI94] of stable semantics for programs admitting (in modern notation) constraints of the form $1X$ in the heads of the clauses. Niemelä, Simons, and Soininen used a modification of the Gelfond-Lifschitz transform [GL88], which we called the NSS transform in [MR02], to define CC-stable models. However, the presence of expressions of the form kXl in cardinality constraint programs forces one to abandon one of fundamental properties of stable models of normal logic programs, namely, that stable models of a normal program are minimal and hence form an antichain with respect to set inclusion. Once atoms of the form kXl are allowed, even CC-programs in which all clauses have empty bodies can have collections of CC-stable models that do not form an antichain and hence not all CC-stable models are minimal. The results of [MR02] show that there is a direct connection between stable models of normal logic programs and CC-stable models of CC-logic programs. That is, CC-stable models are *projections* of a stable models of a suitably chosen normal program in an extended language with a larger set of atoms. This means that the computation of CC-stable models of [NSS99] can be viewed as computing the stable model semantics for that extended program but then one *hides* all the atoms that do not occur in the original program. A similar result, but with respect to a different formalism, the so called answer sets with nested expressions, has been obtained by Ferraris and Lifschitz [FL01].

The main purpose of this paper is to define an extension of CC-logic programs, called *SC-logic programs* where one replaces atoms of the form kXl by a more general set constraint of the form $\langle X, \mathcal{F}\rangle$ where $\mathcal{F}$ is an *arbitrary* family of subsets of X. Here the intended meaning of $\langle X, \mathcal{F}\rangle$ is that in an intended model M, $M \cap X \in \mathcal{F}$. We will call such atoms *set-constraint atoms* or *SC-atoms* and the corresponding programs *SC-logic programs*. It is easy to see that a CC-atom kXl is just a SC-atom $\langle X, \mathcal{F}_{k,l}\rangle$ where $\mathcal{F}_{k,l}$ is a family of subsets of X

$$\mathcal{F}_{k,l} = \{Y \subseteq X : k \leq |Y| \leq l\}.$$

We shall show that results of [NSS99] and the results of [MR02] can be extended to the setting of SC-logic programs. For example, we shall show that one can extend the ideas of [NSS99] to define a natural notion of SC-stable models. Moreover, while the complexity of various problems associated with SC-stable models may be high due to the fact that the Kolmogorov complexity of $\mathcal{F}$ in a SC-atom $\langle X, \mathcal{F}\rangle$ may be large, the basic interpretation result that SC-stable models of SC-programs are traces of stable models of normal logic programs over an extended language continues to hold. We shall show that SC-atoms can incorporate arbitrary aggregation functions and that SC-atoms allow one to express other classes that may be important in applications. We should note that the

SAT community has studied similar issues such as considering *pseudo-Boolean* constraints [ARMS02]. Various applications such as wire outlay on the chips, model checking, and timing of chips motivate the study of such extensions. In the ASP community, more limited extensions of DATALOG$^\neg$ have been studied in [ET01] and other papers.

While there seems to be a natural notion of stable models for SC-programs, the notion of a supported model for SC-programs is not so straightforward. One immediate problem is that the natural extension of the one-step provability operator T_P for a normal logic program P leads to a non-deterministic operator. For example, even for the simple program P which consists of a single clause

$$\langle X, \mathcal{F} \rangle \leftarrow$$

one would naturally define $T_P(A)$ to be $\{Y : Y \in \mathcal{F}\}$ for any A which is a nondeterministic operator. We shall show that one can define a natural notion of SC-supported model for SC-logic programs as a model of P which is a fixed point of an appropriate one-step provability operator. However this approach will force us to investigate nondeterministic operators which have not been previously considered in the logic programming literature. As we shall see the properties of nondeterministic operators are very different from the properties of deterministic operators. For example, there are natural notions of fixed points and monotonicity for nondeterministic operators which reduce to the usual notions of fixed points and monotonicity if the operator is deterministic. However, we shall give an example of a monotone nondeterministic operator that does not have a fixed point. Thus the straightforward generalization of the Tarski-Knaster theorem fails for nondeterministic operators. Instead, we prove two generalizations of Knaster-Tarski theorem that are applicable to nondeterministic operators. One of these generalizations can be applied to show that a SC-stable model of a SC-logic program P is always a SC-supported model.

2 Set Constraints and Logic Programs

Let X be a set. The power set of X, $\mathcal{P}(X)$, is the collection of all subsets of X. A *set constraint atom* for a set X of atoms is a pair $\langle X, \mathcal{F} \rangle$ where $\mathcal{F} \subseteq \mathcal{P}(X)$. Given a set of atoms M and a set constraint atom $\langle X, \mathcal{F} \rangle$, we say that M *satisfies* $\langle X, \mathcal{F} \rangle$, in symbols $M \models \langle X, \mathcal{F} \rangle$, if $M \cap X \in \mathcal{F}$. We say that M satisfies a collection B of set constraint atoms if M satisfies all set constraint atoms in B. A *set constraint clause* (SC-clause for short) is an expression of the form

$$s \leftarrow s_1, \ldots s_k \tag{1}$$

where s and each s_i are set constraint atoms. The body of the clause (1) is the set of set constraint atoms $\{s_1, \ldots, s_k\}$. A *set constraint logic program* (SC-logic program for short) is a collection P of set constraint clauses.

We can now introduce a notion of a *model* of a SC-logic program. A set of atoms M satisfies a clause $C = s \leftarrow s_1, \ldots s_k$ if the fact that all set constraint

atoms in the body of C are satisfied by M implies that M satisfies s as well. A set of atoms M satisfies a SC-logic program P if it satisfies all clauses of P.

We note that the satisfaction of atoms can be easily expressed in terms of the satisfaction of set constraints. Namely, $M \models a$ if and only if $M \models \langle\{a\}, \{\{a\}\}\rangle$. Similarly, satisfaction of negated atoms can also be expressed in terms of the satisfaction of set constraints. Namely, $M \models \neg a$ if and only if $M \models \langle\{a\}, \{\emptyset\}\rangle$. Thus we could write each literal a or $\neg a$ in a normal logic program P as a set constraint atom and hence we can consider each normal logic program as a special case of a SC-logic program. However such a translation makes normal logic programs much harder to read. Thus, in what follows, we shall simply write a for the set constraint atom $\langle\{a\}, \{\{a\}\}\rangle$ and $\neg a$ for the set constraint atom $\langle\{a\}, \{\emptyset\}\rangle$.

As we mentioned in the introduction, set constraint atoms can express general aggregation functions.

Example 1. (**Cardinality and Weight Constraint Atoms**) As described in the introduction a CC-atom kXl can be expressed as the SC-atom $\langle X, \mathcal{F}_{k,l}\rangle$ where $\mathcal{F}_{k,l} = \{Y \subseteq X : k \leq |Y| \leq l\}$. Similarly if we have a weight function wt on literals, the more general weight constraint kXl considered [NS00] where a model M satisfies kXl if and only if

$$k \leq [\sum_{a \in X \cap M} wt(a) + \sum_{b \in X - M} wt(\neg b)] \leq l$$

can be expressed as the SC-atom $\langle X, \mathcal{F}\rangle$ where

$$\mathcal{F} = \{Y \subseteq X : k \leq [\sum_{a \in Y} wt(a) + \sum_{b \in X - Y} wt(\neg b)] \leq l\}.$$

□

Example 2. (**SQL Aggregate Atoms**) Let X be a finite set of atoms and let $\mu : X \to \mathrm{R}$ be a real function. Each such function μ allows us to construct a variety of set constraint atoms. For example, to each $Y \subseteq X$, we can assign the following functions that are used in SQL queries: $|Y|$, $\mathrm{sum}(Y) = \sum_{y \in Y} \mu(y)$, $\min(Y) = \min_{y \in Y} \mu(y)$, $\max(Y) = \max_{y \in Y} \mu(y)$, $\mathrm{avg}(Y)$, where avg assigns to Y the real number 0 if $Y = \emptyset$ and assigns the real number $\frac{\mathrm{sum}(Y)}{|Y|}$, otherwise. For every two real numbers a, b such that $a \leq b$, we define the following families of sets:

1. $C_X^{a,b} = \{Y : a \leq |Y| \leq b\}$
2. $S_X^{a,b} = \{Y : a \leq \mathrm{sum}(Y) \leq b\}$
3. $\mathrm{Max}_X^{a,b} = \{Y : a \leq \max(Y) \leq b\}$
4. $\mathrm{Min}_X^{a,b} = \{Y : a \leq \min(Y) \leq b\}$
5. $\mathrm{avg}_X^{a,b} = \{Y : a \leq \mathrm{avg}(Y) \leq b\}$

For each family $\mathcal{F}$ described in (1)-(5), we obtain a set constraint $\langle X, \mathcal{F}\rangle$. □

Example 3. (**Programs with External Modules**) In [EGV97], Eiter, Gottlob and Veith studied logic programs whose clauses contain *modules* in their bodies. Modules are programs π (written in some fixed programming language) that return subsets of some finite set of atoms X. Let us define R_π as the set of those subsets of X that can be returned by π. Eiter, Gottlob and Veight show how a stable semantics can be assigned to programs that contain atoms of the form $\langle X, R_\pi \rangle$ in the body of clauses. Our construction of SC-stable models below extends the work of [EGV97] in that SC-logic programming allows modules to occur both in the heads and in the bodies of clauses. □

It should be however clear, that there are other families of subsets of a set that are of interest.

Example 4. Given a finite set of atoms, let $\mathcal{F}_{even} = \{Y \subseteq X : |Y| \textit{ is even}\}$ and $\mathcal{F}_{odd} = \{Y \subseteq X : |Y| \textit{ is odd}\}$. Then $\langle X, \mathcal{F}_{even} \rangle$ and $\langle X, \mathcal{F}_{odd} \rangle$ are set constraint atoms. □

Clearly, the notion of satisfaction defined above generalizes the usual notion of satisfaction for Horn logic programming clauses and programs. Unlike Horn logic programs, SC-logic programs do not have to have models even in the case where the body of each SC-clause in empty.

Example 5. Consider the SC-logic program P which consisting of the following two clauses. $\langle \{a, b\}, \mathcal{F}_{Even} \rangle \leftarrow \quad \langle \{a, b\}, \mathcal{F}_{Odd} \rangle \leftarrow$. It is easy to see that P has no model M since to be a model of P would require that $|M \cap \{a, b\}|$ is both even and odd. □

We can, however, prove the following.

Proposition 1. *If a SC-logic program P possesses a model, then it possesses an inclusion-minimal model.*

3 Stable Models of Set Constraint Logic Programs

In this section, we shall generalize the notion of CC-stable models introduced by Niemelä, Simons and Soininen [NSS99] to the class of SC-logic programs. To understand our extension, we first formally define cardinality constraint logic programs (CC-logic programs). The syntax of CC-logic programs admits two types of atoms: (i) ordinary atoms from a set At and (ii) atoms of the form kXl where X is a finite set of atoms from At, k is a natural number (i.e. $k \in \omega$), $l \in \omega \cup \{\infty\}$ and $k \leq l$. When $l = \infty$, we abbreviate kXl as kX. Such new atoms will be called *cardinality constraints*. The intended meaning of an atom kXl is "out of atoms in X at least k but not more than l belong to the intended model." Notice that the meaning of the negated atom, $\neg p$ is precisely the same as that of $0\{p\}0$. Therefore we shall assume that the bodies of rules of CC-logic programs contain only atoms of the form kXl and atoms from At. That is, a CC-clause is either a clause of the form

$$p \leftarrow q_1, \ldots, q_m, k_1 X_1 l_1, \ldots, k_n X_n l_n \tag{2}$$

or

$$kXl \leftarrow q_1, \ldots, q_m, k_1X_1l_1, \ldots, k_nX_nl_n. \tag{3}$$

We note that either m or n can be zero. Thus the head of CC-clauses is either of the form p where p is an atom from At or kXl where k, X, and l satisfy the conventions described above. We say that a set of atoms $M \subseteq At$ satisfies the cardinality constraint kXl, in symbols $M \models kXl$, if $k \leq |X \cap M| \leq l$. Similarly we say that $M \models p$ where $p \in At$, if $p \in M$. By treating the commas in the bodies of clauses as conjunctions, we say that $M \models body(C)$ if all atoms occurring in $body(C)$ belong to M and all cardinality constraints occurring in $body(C)$ are satisfied by M. Finally, we say that M satisfies a clause C, $M \models C$, if either M does not satisfy the body of C or M satisfies the head of C.

A CC-logic program is a set of CC-clauses of the form (2) or (3). We say that M is model of P, $M \models P$, if M satisfies all CC-clauses $C \in P$.

A class of programs called Horn CC-programs play a role similar to that of Horn programs in ordinary logic programming. A *Horn CC-clause* is a CC-clause where the head of the clause is an ordinary atom and all the cardinality constraint atoms $k_iX_il_i$ in the body have $l_i = \infty$, i.e., it is of the form

$$H = p \leftarrow q_1, \ldots q_m, k_1X_1, \ldots, k_nX_n$$

Niemelä, Simons and Soininen observe that the one-step provability operator associated with a Horn CC-program is monotone and hence a Horn CC-program P has a least fixed point, M^P. Moreover, they show that M^P is the least model of P.

Next we introduce the analogue of the Gelfond-Lifschitz reduct for CC-logic logic programs which we call the NSS-reduct. The NSS-reduct of a CC-logic program P with respect to a set M of ordinary atoms is defined as follows. First we eliminate all clauses D of P such that M does not satisfy the body of D. For the remaining clauses C of P, replace C by C^M where

1. $C^M = p \leftarrow q_1, \ldots, q_m, k_1X_1 \ldots, k_nx_n$ if $C = p \leftarrow q_1, \ldots, q_m, k_1X_1l_1, \ldots, k_nX_nl_n$ and

2. C^M is a collection of Horn constraint clauses of the form $p \leftarrow q_1, \ldots, q_m, k_1X_1, \ \ldots, k_nX_n$ for each $p \in X \cap M$ if $C = kXl \leftarrow q_1, \ldots, q_m, k_1X_1l_1, \ \ldots, k_nX_nl_n$.

We let P^M denote the Horn CC-program consisting of the set of all C^M such that $C \in P$ and M satisfies the body of C. Following [NSS99], we say that M is a *CC-stable model* of P if (i) M is a model of P and (ii) M is the least model of the Horn CC-program P^M.

To define the notion of a SC-stable models for a SC-logic program, we first must define an extension of the NSS-reduct. Our first step is to define an appropriate analogue of clauses of the form $kX\infty$. To this end, define the upper-closure of $\mathcal{F}$ with respect to X to be the family $\overline{\mathcal{F}}_X$ where

$$\overline{\mathcal{F}}_X = \{Y \subseteq X : \exists_Z(Z \in \mathcal{F} \wedge Z \subseteq Y)\}.$$

We will drop the subscript X when it is determined by the context. A family of subsets of X is *closed* if $\overline{\mathcal{F}} = \mathcal{F}$. A closed family is nothing more than an upper ideal in the partially ordered set $\langle \mathcal{P}(X), \subseteq \rangle$. Notice that closure of a closed family $\mathcal{F}$ of subsets of X is $\mathcal{F}$ itself.

Example 6. The closure kXl is $kX\infty$ (that is, kX), i.e. family $\{Y \subseteq X : k \leq |Y|\}$. The closure of $\mathcal{F}_{even}$ is the entire powerset of X (recall that X is finite). The closure of $\mathcal{F}_{odd}$ is the set of all non-empty subsets of X.

Observe that an atom a is shorthand for the SC-atom $\langle \{a\}, \{\{a\}\} \rangle$ which is automatically closed. However the atom $\neg a$ is shorthand for the SC-atom $\langle \{a\}, \{\emptyset\} \rangle$ whose closure is $\langle \{a\}, \{\emptyset, \{a\}\} \rangle$. □

Clearly, different families of sets may generate the same closure. However, closure of a family $\mathcal{F}$, $\overline{\mathcal{F}}$, has precisely the same inclusion-minimal elements as $\mathcal{F}$. We define the *closure* of an SC-atom $\langle X, \mathcal{F} \rangle$ to be $\langle X, \overline{\mathcal{F}} \rangle$.

This given, we can now define the analogue of Horn program. A *Horn SC-clause* is a SC-clause where the head of the clause is an ordinary atom and all SC-atoms in the body are closed, i.e. a clause of the form

$$H = p \leftarrow q_1, \ldots q_m, \langle X_1, \mathcal{F}_1 \rangle, \ldots, \langle X_n, \mathcal{F}_m \rangle.$$

where $\mathcal{F}_i = \overline{\mathcal{F}_i}$ for all i. A Horn SC-logic program is a SC-program consisting entirely of Horn SC-clauses. As in [NSS99], one can show that the one-step provability operator associated with a Horn SC-program is monotone and hence a Horn SC-program P has a least fixed point, M^P, which is a unique minimal model of P. Thus we have the following.

Proposition 2. *Let P be a Horn SC-logic program. Then:*

1. *There is a least model of P, M_P.*
2. *There is a deterministic monotone operator S_P such that M_P is the least fixed point of S_P. The fixed point of S_P is reached in at most ω steps regardless of the size of P.*

We will now define the NSS transform of a SC-logic program P with respect to a set of atoms M. Let P be a SC-logic program and let M be a subset of At. The NSS transform , $\mathrm{NSS}(P, M)$, of P with respect to M is defined in two steps. First, eliminate from P all clauses whose bodies are *not* satisfied by M. In the second step, in each remaining clause we execute the same operations as in the original NSS transform, except that the closure of the atoms in the bodies of clauses is as defined above. That is for each clause $\langle X, \mathcal{F} \rangle \leftarrow q_1, \ldots, q_m, \langle X_1, \mathcal{F}_1 \rangle, \ldots, \langle X_k, \mathcal{F}_k \rangle$, and for each $a \in X \cap M$, we generate the clause $a \leftarrow q_1, \ldots, q_m, \langle X_1, \overline{\mathcal{F}_1} \rangle, \ldots, \langle X_k, \overline{\mathcal{F}_k} \rangle$.

It is easy to see that the resulting program $\mathrm{NSS}(P, M)$ is a Horn SC-logic program. Consequently, $\mathrm{NSS}(P, M)$ has a least model $N_{P,M}$. We then say that M is an *SC-stable model of* P if (I) M is a model of P and (II) $M = N_{P,M}$.

We note that the first condition that M is model of P need not to be required for normal logic program P because the least model of the Gelfond-Lifschitz transform of P is automatically a model of P. This is not true for SC-logic programs as our next example will show.

Example 7. Let P be a SC-logic program consisting of the following two clauses:

$$1\{a,b,c,d\}2 \leftarrow \quad 3\{a,b,c,d\}4 \leftarrow$$

Note that $M = \{a,b,c,d\}$ is not a model of P. In fact, it is easy to see that P has no models. However $\mathrm{NSS}(P,M)$ consists of four atomic clauses:
$a \leftarrow \quad b \leftarrow \quad c \leftarrow \quad d \leftarrow$. Thus $M = N_{P,M}$. However, since M is *not* a model of P, M is not a stable model of P. □

We note that pruning process for the NSS-transform is more extensive than in the case of Gelfond-Lifschitz (GL) transform of normal logic programs. That is, for the GL-transform, we prune those clauses where M contradicts the negative part of the body, while for the NSS-transform, we eliminate clauses with bodies *not* satisfied by M. As shown by Truszczynski [MT95], this stronger Gelfond-Lifschitz-like transformation leads to the same class of stable models.

Let P be a normal logic program. We identify P with a SC-logic program where each atom and each negated atom are expressed by set constraints (a is expressed by $\langle\{a\},\{\{a\}\}\rangle$, $\neg a$ is expressed by $\langle\{a\},\{\emptyset\}\rangle$). We then have the following result which is essentially the same result that Niemelä, Simons and Soininen established for CC-logic programs.

Proposition 3. *Let P be a normal logic program and let M be a set of atoms. Then M is a stable model of P in the sense of Gelfond and Lifschitz if and only if M is a stable model of P viewed as a SC-logic program.*

We end this section with an analogue of the main result of [MR02] for SC-logic programs.

Theorem 1. *Let P be a SC-logic program over a language $\mathcal{L}$. Then there is a normal logic program $\overline{P}$ over a extended language $\overline{\mathcal{L}}$ of $\mathcal{L}$ such that*

(i) *For each SC-stable model M of P, there is a unique stable model $\overline{M}$ of $\overline{P}$ such that M is the restriction of $\overline{M}$ to the language $\mathcal{L}$.*
(ii) *For each stable model $\overline{M}$ of $\overline{P}$, the restriction of $\overline{M}$ to the language $\mathcal{L}$ is a SC-stable model of P.*

4 Nondeterministic Lattice Operators

We observed that introduction of set constraints leads naturally to investigation of non-deterministic operators in complete lattices. Consequently, in this section, we will investigate nondeterministic operators and establish some of their properties.

Let $\langle L, \leq_L\rangle$ be a complete lattice. A *(nondeterministic) operator* in L is any function O from L to the powerset of L. We say that an operator O is *deterministic*, if for every $x \in L$, the size of $O(x)$, $|O(x)|$, is equal to 1. If the operator O is deterministic, we can identify O with a mapping from L to L, namely, assigning to $x \in L$ the unique element of $O(x)$. Conversely, a mapping

Q from L to L can be identified with a nondeterministic operator O_Q from L to $\mathcal{P}(L)$ by assigning to every x the set consisting of a single lattice element $Q(x)$.

Nondeterministic operators naturally occur in the context of set constraint programs as our next example will show.

Example 8. Let P consist of a single clause

$$1\{p_1, p_3\}2 \leftarrow 2\{p_2, p_4, p_5\}3$$

It is natural to assign to $M = \{p_1, p_2, p_5\}$ each of the following values $\{p_1\}$, $\{p_3\}$, and $\{p_1, p_3\}$. Unless additional criteria are used, each of these values is a correct value because the intention of the programmer described in this clause is that *any* of these values is acceptable. □

We say that a nondeterministic operator $O : L \to \mathcal{P}(L)$ is *monotone* if for every $x, y \in L$ such that $x \leq_L y$, it is the case that for every $z \in O(x)$, there is $t \in O(y)$ such that $z \leq_L t$. A fixed point of an operator $O : L \to \mathcal{P}(L)$ is any x such that $x \in O(x)$.

Proposition 4. *If $O : L \to L$ is a monotone operator, then its interpretation as a nondeterministic operator from L to $\mathcal{P}(L)$ is also monotone.*

Thus a monotone *deterministic* operator always possesses a fixed point. It is tempting to conjecture that a nondeterministic monotone operator always possesses a fixed point. However, this is not always the case as our next example will show.

Example 9. Let L be the Boolean lattice $\mathcal{P}(N)$ where N is the set of non-negative integers. The structure $\langle L, \subseteq \rangle$ forms a complete lattice. Define a nondeterministic operator O from $\mathcal{P}(N)$ to $\mathcal{P}(\mathcal{P}(N))$ as follows.

(a) If $X \subseteq N$ is finite set of cardinality n, then $O(X)$ is a family consisting of a single set $\{0, \ldots, n\}$.
(b) If $X \subseteq N$ is an infinite set, then $O(X)$ is the family of all finite subsets of N, $\mathcal{P}_{fin}(N)$.

First, observe that O is a monotone nondeterministic operator. Indeed, assume $X \subseteq Y$.
Case 1. Both X and Y are finite. If $X \subseteq Y$ so that $|X| \leq |Y|$, then $O(X) = \{\{0, \ldots, |X|\}\}$, $O(Y) = \{\{0, \ldots, |Y|\}\}$ and every element of $O(X)$ is contained in every element of $O(Y)$ and hence the monotonicity condition holds.
Case 2. Both X and Y are infinite. Then every element of $O(X)$ belongs to $O(Y)$, thus the monotonicity condition holds.
Case 3. X is finite and Y is infinite. Then since $O(Y) = \mathcal{P}_{fin}(N)$, the only element of $O(X)$, being finite, belongs to $O(Y)$. Thus again the monotonicity condition holds.

However, O has no fixed point. For let X be a subset of N. If X is finite, then the only element of $O(X)$ has size bigger than that of X, and thus X cannot be a fixed point. When X is infinite, X does not belong to $O(X)$ at all. □

A close look at Example 9 will show that there are two reasons for the lack of fixed points. First, we allowed $O(X)$ be infinite when X is infinite. Second, the limit of the values of O of a directed family is not a value of O. We will now state two results on the existence of fixed points of monotonic nondeterministic operators.

Proposition 5. *Let O be a monotone nondeterministic operator from a complete lattice L to $\mathcal{P}(L)$ such that $O(\bot)$ is nonempty, and for every X, $O(X)$ is finite. Then O possesses a fixed point.*

Proposition 6. *Assume that L is a complete lattice and $O : L \to \mathcal{P}(L)$ is a nondeterministic operator satisfying following two properties:*

1. *O is monotone*
2. *For every $x \in L$, the family $O(x)$ has a maximum element*

Then O possesses a fixed point.

Notice that both Propositions 5 and 6 are generalizations of Knaster-Tarski theorem in that monotone deterministic operators automatically satisfy the hypotheses of the each theorem.

5 Generalization of van Emden-Kowalski Operator

In this section, we develop an analogue of the one-step provability operator for SC-logic programs. This operator is a generalization of the familiar van Emden Kowalski operator [AvE82].

Let P be a SC-logic program. If M be a set of atoms, then we let $Sat_{P,M} = \{C \in P : M \models body(C)\}$. Thus $Sat_{P,M}$ consists of those SC-clauses in P such that M satisfies the body of C. Fix a SC-logic program P. An *M-satisfier* is any function from $Sat_{P,M}$ to $\mathcal{P}(At)$ which assigns to each clause $C \in Sat_{P,M}$, an element of the family $\mathcal{F}$ for which $\langle X, \mathcal{F}\rangle$ is the head of C. Thus an M-satisfier provides values to satisfy the heads of clauses whose bodies are satisfied by M.

Example 10. Let P be this SC-logic program:

$C_1 :\ \langle\{a, b, c\}, \mathcal{F}_{even}\rangle \leftarrow a$
$C_2 :\ 2\{a, b, c, d\}3 \leftarrow 1\{b, c, d\}3$
$C_3 :\ c \leftarrow b$

Take as M the set $\{b, d\}$. It satisfies the second and the third clauses, but not the first one. There are several M-satisfiers for M. Clearly every M-satisfier must assign $\{c\}$ to clause C_3. However for clause C_2, an M-satisfier can assign any two or three element subset of $\{a, b, c, d\}$. □

Now, we are ready to define the nondeterministic operator T_P associated with the program P. Specifically, we define

$$T_P(M) = \{\bigcup Rng(f) : f \text{ is an } M\text{-satisfier}\}$$

Thus the value of the T_P operator on M is the collection of "candidates", each candidate being the union of the range of some M-satisfier.

Example 11. In our previous example there are 7 possible values for $T_P(\{b, d\})$. Each of those contains c. When we inspect $N = \{b, c, d\}$, we again find the same 7 values in $T_P(N)$. It is easy to see that the set N is a fixed point for the operator T_P. □

We note that the nondeterministic operator T_P for SC-logic programs P is, in fact, a generalization of the familiar van Emden-Kowalski operator. Indeed, if P is a normal logic program and M is a subset of the set of atoms At, then there is just one M-satisfier. Specifically, it is a unique function $f : Sat_{P,M} \to \mathcal{P}(At)$ such that $f(C) = \{head(C)\}$ where $head(C)$ is the head of C. Thus for normal logic programs, the operator T_P is deterministic.

Recall that an atom $\langle X, \mathcal{F}\rangle$ *closed*, if $\mathcal{F}$ is an upper ideal in $\mathcal{P}(X)$, that is, if

$$\forall_{Y,Z}(Y \in \mathcal{F} \wedge Y \subseteq Z \subseteq X \Rightarrow Z \in \mathcal{F}).$$

An SC-logic program P is *closed* if all the SC-atoms which occur either in the head or the body of a clause P are closed. Observe that if the atom $\langle X, \mathcal{F}\rangle$ is closed, $M \models \langle X, \mathcal{F}\rangle$ and $M \subseteq M'$, then $M' \models \langle X, \mathcal{F}\rangle$. In fact, this property characterizes the closed SC-atoms. Specifically, if for all $M \subseteq M' \subseteq At$, $M \models \langle X, \mathcal{F}\rangle$ implies $M' \models \langle X, \mathcal{F}\rangle$, then $\mathcal{F}$ must be closed.

We now have the following result.

Proposition 7. *If P is a closed SC-logic program, then T_P possesses a fixed point M. Moreover M can be chosen to be a model of P.*

Unlike the situation for normal logic programs, it is not the case that every fixed point of T_P is a model of P as our next example will show.

Example 12. Let P be this program:

$1\{p, q, r\}2 \leftarrow p$
$2\{p, q, r\}3 \leftarrow p$

This program has four fixed points: $\emptyset$, $\{p, q\}$, and $\{p, r\}$ and $\{p, q, r\}$. It is easy to see that first three are models of P while the last one is not.

Clearly, it is even easier to find a model which is not a fixed point. Any non-supported model of a normal logic program induces such example.

We call a set M of atoms a *supported* model of a SC-logic program P if M is a model of P and M is a fixed point of the nondeterministic operator T_P.

Our final result of this section shows that stable models of SC-logic programs are fixed points of the nondeterministic operator T_P considered in Section 5. This generalizes the result of Gelfond and Lifschitz which is that stable models of normal logic programs are always supported models.

Proposition 8. *Let P be a SC-logic program and let M be a set of atoms. If M is a stable model of P then M is a fixed point of the nondeterministic operator T_P.*

6 Conclusions and Further Research

We discuss some issues related to the research presented in this paper. The first question is: "What families are representable as families of stable models of a program?" In the case of a finite collection of finite sets, the answer is obvious. Each such family $\mathcal{X}$ is representable. Indeed, if $\mathcal{X}$ is empty, then any inconsistent program is appropriate. If $\mathcal{X}$ is nonempty, then two cases are possible. If $\mathcal{X}$ consists of an empty set, then $\neg p \leftarrow$ is the appropriate program. Otherwise, take $X = \bigcup \mathcal{X}$ and let P consist of a single clause $\langle X, \mathcal{X} \rangle \leftarrow$. It is easy to see that P can be used to represent $\mathcal{F}$.

It turns out that the assumption that all sets in the family $\mathcal{F}$ are finite is immaterial. That is, we can prove if $\mathcal{F}$ is any finite collection of sets of atoms, then there is a SC-logic program $P_{\mathcal{F}}$ such that the family of SC-stable models of $P_{\mathcal{F}}$ is $\mathcal{F}$.

Unfortunately, these results tell us nothing about what infinite families sets can form the set of SC-stable models of some SC-logic program. In the case stable models of normal logic programs, two characterizations are available. A topological characterization of the families representable as the set of stable modes of a logic program has been found by A. Ferry [Fe94]. An alternative solution, in recursion-theoretic terms, has been found in [MNR90]. No corresponding result for SC-logic programs, or even CC-logic programs, are known.

Another problem which awaits settlement is the problem of well-founded semantics [VRS91] for SC-logic programs. It is clear that *some* form of well-founded semantics can be obtained by the reduction to the stable semantics of normal logic program. We believe, though, that a more direct construction via approximation of the nondeterministic operator in the spirit of [DMT02] exists.

Acknowledgments. The first author's research has been partially supported by the NSF grants IIS-0097278 and IIS-0325063. The second author's research has been partially supported by the ARO contract DAAD19-01-1-0724.

References

[ARMS02] F.A. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack-search pseudo-boolean solver and optimizer. SAT02, 2002.

[AvE82] K.R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[CMT96] P. Cholewiński, W. Marek, and M. Truszczyński. Default reasoning system DeReS. KR96, pages 518–528. Morgan Kaufmann, 1996.

[ASP01] Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming, Stanford, CA, USA, 2001.

[Ba03] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, 2003.

[DMT02] M. Denecker, V.W. Marek and M. Truszczyński. Ultimate approximations in nonmonotonic knowledge representation systems. KR02, pages 177–188, Morgan-Kaufmann, 2002.

[ET01] D. East and M. Truszczynski. More on wire-routing. In: [ASP01], 2001.

[ET02] D. East and M. Truszczynski. *aspps* solver. `http://www.cs.uky.edu/ai/`. 2002.

[EGV97] T. Eiter, G. Gottlob and H. Veith. Modular Logic Programs and General Quantifiers, LPNMR97, pages 290–309, 1997.

[ELM+98] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System dlv: Progress Report, Comparisons, and Benchmarks. KR98, pages 406–417, 1998.

[FL01] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. To appear in *Theory and Practice of Logic Programming*.

[Fe94] A. Ferry. *Topological Characterizations for Logic Programming Semantics*, Ph.D. Dissertation, University of Michigan, 1994.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. ISLP88, pages 1070–1080, 1988.

[KS99] H.A. Kautz and B. Selman. Unifying sat-based and graph-based planning. IJCAI99, pages 318–325. Morgan Kaufmann, 1999.

[MNR90] V. Marek, A. Nerode, and J. B. Remmel. Nonmonotonic rule systems I. *Annals of Mathematics and Artificial Intelligence*, 1: 241–273, 1990.

[MR02] V. W. Marek and J.B. Remmel. On logic programs with cardinality constraints. NMR9, pages 219–228, 2002.

[MT95] V. Marek and M. Truszczyński. Revision Programming. *Theoretical Computer Science* 190(2):241–277, 1995.

[MT99] V. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. *The Logic Programming Paradigm*, pages 375–398. Springer-Verlag, 1999.

[NS97] I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs, LPNMR97, pages 420–429, 1997.

[NS00] I. Niemelä and P. Simons. Extending Smodels System with Cardinality and Weight Constraints. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.

[NSS99] I. Niemelä, P. Simons. and T. Soininen. Stable Model Semantics of Weight Constraint Rules. LPNMR99, pages 317–331, 1999.

[SI94] C. Sakama and K. Inoue. An alternative approach to the Semantics of Disjunctive Logic Programs and Deductive Databases. *Journal of Automated Reasoning* 13:145–172. 1994.

[SK92] B. Selman and H. A. Kautz. Planning as satisfiability. ECAI-92, pages 359–363. Wiley, 1992.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38:620–650, 1991.

Verifying the Equivalence of Logic Programs in the Disjunctive Case*

Emilia Oikarinen and Tomi Janhunen

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
{Emilia.Oikarinen,Tomi.Janhunen}@hut.fi

Abstract. In this paper, we address the problem of checking whether two disjunctive logic programs possess exactly the same stable models. An existing translation-based method [14], which was designed for weight constraint programs supported by the SMODELS system, is generalized to the disjunctive case. Moreover, we report on our preliminary experiments with an implementation of the method, a translator called DLPEQ.

1 Introduction

Answer set programming (ASP) is a logic programming paradigm [21,22] where a problem at hand is solved in a declarative fashion by (i) writing a logic program whose *answer sets* correspond to the solutions of the problem; and by (ii) computing the answer sets of the program using a special purpose search engine. There is a growing interest towards ASP and much of that is due to efficient search engines such as DLV [15], SMODELS [28], and GNT [12,13] available today. Consequently, a variety of interesting applications of ASP has emerged. Let us just mention a decision support system of the space shuttle [1] as a fine example.

The standard syntax of logic programs is that of *normal* or *general* logic programs [19], which admits the use of Clark's *negation as failure* principle in the bodies of rules. A number of generalizations to the basic syntax have been proposed: classical negation [6], disjunction in the heads of rules [7,27], default negation in the heads of disjunctive rules [9], nested rules [17], and weight rules [28]. The semantics of the resulting classes of logic programs is determined by the respective generalizations of the *stable model semantics* [5] which has settled as a leading semantics in ASP. According to [6], the *answer sets* mentioned above are consistent sets of classical literals that are stable in the same sense [5] as stable models are. In the sequel, we concentrate on disjunctive logic programs, which cover the kinds of programs mentioned above; either as a proper generalization or via suitable translations [4,6,9,10,25]. Thus a wide variety of logic programs is implicitly under consideration in this paper.

* The research reported in this paper is partially funded by the Academy of Finland under the project "Applications of Rule-Based Constraint Programming" (#53695).

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 180–193, 2004.

Despite the declarative nature of ASP, the development of programs resembles that of programs in conventional programming. That is, a programmer often develops a series of improved programs for a particular problem, e.g., when optimizing the execution time and space. As a consequence, the programmer needs to ensure that subsequent programs which differ in performance yield the same output. This leads us to the problem of checking whether given two logic programs P and Q give rise exactly to the same answer sets. If this is the case, then P and Q are called *(weakly) equivalent*, denoted by $P \equiv Q$. Note that such a notion of equivalence can be considered as a sufficient one from the programming perspective, as answer sets capture the solutions of the problem being solved.

In [14], we develop a method for testing the (weak) equivalence of *weight constraint programs* supported by the SMODELS system [28]. The key idea in this approach is to *translate* logic programs P and Q under consideration into a single logic program $\mathrm{EQT}(P, Q)$ which has an answer set if and only if P has an answer set that is not an answer set of Q. Such answer sets, if found, act as *counter-examples* to the equivalence of P and Q. Consequently, the equivalence of P and Q can be established by showing that $\mathrm{EQT}(P, Q)$ and $\mathrm{EQT}(Q, P)$ have no answer sets.[1] Thus the existing search engine SMODELS can be used for the search of counter-examples and there is no need to develop a special purpose search engine for equivalence testing. Only a translator like LPEQ [11] needs to be implemented. The aim of the current paper is to generalize the translation-based method to the disjunctive case so that search engines dedicated to disjunctive logic programs such as GNT [12] can be used for actual computations.

There are also other notions of equivalence that have been proposed for logic programs. Lifschitz et al. [16] consider P and Q *strongly equivalent*, denoted $P \equiv_\mathrm{s} Q$, iff P and Q yield exactly the same answer sets in any *context*, i.e. as parts of another program R. The resulting equivalence relation is fairly strong and better applicable to *subprograms* or *program modules* constituting larger programs rather than complete programs. Note that $P \equiv_\mathrm{s} Q$ implies $P \equiv Q$ by setting $R = \emptyset$ above, but the converse implication does not hold in general. Consequently, the question whether $P \equiv Q$ holds remains open whenever $P \not\equiv_\mathrm{s} Q$ turns out to be the case. This implies that verifying $P \equiv Q$ remains as a problem of its own, which cannot be fully compensated by verifying $P \equiv_\mathrm{s} Q$. This view is supported by the complexity results: deciding $\equiv$ for finite propositional disjunctive programs $\Pi_2^\mathbf{P}$-hard [29] whereas deciding $\equiv_\mathrm{s}$ is only **coNP**-complete [26,18]. Thus, unless the polynomial hierarchy collapses, it is intractable in general to establish $P \equiv Q$ by choosing certain subprograms P' and Q' of P and Q, respectively, and verifying $P' \equiv_\mathrm{s} Q'$. Moreover, the difference in computational time complexity suggests that weak equivalence is better suited for disjunctive solvers, which are able to decide properties in $\Sigma_2^\mathbf{P}$, whereas SAT solvers and SMODELS suffice for the verification of strong equivalence [18]. For the reasons above, we concentrate on complete programs and the verification of weak equivalence in this paper. Quite recently, Eiter and Fink [2] address yet another notion of equiv-

[1] Turner [29] develops an analogous transformation for weight constraint programs, but for another notion of equivalence, namely *strong equivalence* discussed below.

alence, namely *uniform equivalence* $\equiv_{\mathrm{u}}$, which is a variant of $\equiv_{\mathrm{s}}$ based on sets of facts as possible contexts. However, the problem of deciding $\equiv_{\mathrm{u}}$ is $\Pi_2^{\mathbf{P}}$-complete, suggesting another generalization of the translation-based method.

We proceed as follows. In Section 2, we briefly review the syntax and semantics of disjunctive logic programs. The formal definitions of the equivalence relations for disjunctive logic programs are given in Section 3. Moreover, we discuss the computational cost of the equivalence testing problem. Then, we are ready to present two alternative translation-based techniques for testing the equivalence of disjunctive programs in Section 4. We also address the correctness of the translations. Section 5 shows some results from our preliminary experiments with a prototype implementation called DLPEQ [23] and a solver GNT [12] for disjunctive programs. Section 6 concludes the paper.

2 Disjunctive Logic Programs

We use the symbol "$\sim$" to denote *negation as failure to prove* or *default negation* to distinguish it from classical negation $\neg$. *Default literals* are either atoms a or their default negations $\sim a$. Given a set of atoms A, we define a set of negative literals $\sim A = \{\sim a \mid a \in A\}$. A propositional *disjunctive logic program* (DLP)[2] P is a set of *rules* which are expressions of the form

$$a_1 \mid \ldots \mid a_n \leftarrow b_1, \ldots, b_m, \sim c_1, \ldots, \sim c_k, \tag{1}$$

where $a_1, \ldots, a_n$, $b_1, \ldots, b_m$, and $c_1, \ldots, c_k$ are propositional atoms and n, m and k are nonnegative integers. The *head* of the rule $a_1 \mid \ldots \mid a_n$ is interpreted disjunctively while the rest forming the *body* of the rule is interpreted conjunctively. Intuitively, the rules of the form (1) are used as inference rules: any of the head atoms $a_1, \ldots, a_n$ can be inferred given that the positive body atoms $b_1, \ldots, b_m$ can be inferred and the negative body atoms $c_1, \ldots, c_k$ cannot. Since the order of atoms is not significant in a rule (1) we use a shorthand $A \leftarrow B, \sim C$ where A, B and C are the respective sets of atoms. Rules are separated with full stops and we drop the symbol "$\leftarrow$" if $m = k = 0$. An empty head is denoted by "$\bot$" and a rule with an empty head is called an *integrity constraint*. A program P is *disjunction-free* or a *normal program* if $n = 1$ for each rule of P. Similarly, each rule (1) of a *positive program* P is supposed to satisfy $k = 0$.

Let us then turn our attention to the semantics of a DLP P. The *Herbrand base* $\mathrm{Hb}(P)$ is defined as the set of atoms appearing in P. An *interpretation* $I \subseteq \mathrm{Hb}(P)$ of P defines which atoms $a \in \mathrm{Hb}(P)$ are true ($a \in I$) and which are false ($a \notin I$). An interpretation I is a (classical) *model* of P, denoted by $I \models P$, if and only if for every rule $A \leftarrow B, \sim C$ of P, $B \subseteq I$ and $C \cap I = \emptyset$ imply $A \cap I \neq \emptyset$. For a *minimal model* M of P, there is no other interpretation $M' \subset M$ such that $M' \models P$. The set of minimal models of P is denoted by $\mathbf{MM}(P)$. If P is positive, then $\mathbf{MM}(P)$ gives the standard minimal model semantics of P. If P is also normal, then P has a unique minimal model, i.e. $|\mathbf{MM}(P)| = 1$ [19].

[2] DLPs with variables can be covered by performing a Herbrand instantiation.

However, minimal models do not properly capture the intuitions behind DLPs involving default negation. E.g., $\mathbf{MM}(P_1) = \mathbf{MM}(P_2) = \{\{a\}, \{b\}\}$ for $P_1 = \{a \mid b\}$ and $P_2 = \{a \leftarrow \sim b\}$, although only the first model seems intuitive for P_2 lacking a rule for b. The problem can be resolved by the notion of a *stable model* first proposed for normal programs [5] and then generalized for DLPs [27,7].

Definition 1. *Given a DLP P and an interpretation $M \subseteq \mathrm{Hb}(P)$, the Gelfond-Lifschitz reduct of P is $P_M = \{A \leftarrow B \mid A \leftarrow B, \sim C \in P \text{ and } M \cap C = \emptyset\}$ and M is a stable model of P if and only if $M \in \mathbf{MM}(P_M)$.*

3 Equivalence Testing

Lifschitz et al. [16] recently introduced two notions of equivalence for *nested logic programs* that form a proper generalization of DLPs. In this paper, we define these notions of equivalence for DLPs. We denote the set of stable models of a DLP P by $\mathbf{SM}(P)$. The first notion of equivalence follows naturally from the stable model semantics. DLPs P and Q are equivalent ($P \equiv Q$) iff $\mathbf{SM}(P) = \mathbf{SM}(Q)$. The second notion is defined in terms of the first. DLPs P and Q are strongly equivalent ($P \equiv_s Q$) iff $P \cup R \equiv Q \cup R$ for all DLPs R. The DLP R in definition of $\equiv_s$ can be seen as an arbitrary context for P and Q. Thus P and Q are strongly equivalent iff they have the same stable models in every context in which they can be placed. Clearly, $P \equiv_s Q$ implies $P \equiv Q$, but not vice versa.

Example 1. Consider programs $P = \{a \leftarrow \sim c\}$ and $Q = \{a \leftarrow \sim b\}$. P and Q have the same stable models, i.e. $\mathbf{SM}(P) = \{\{a\}\} = \mathbf{SM}(Q)$, and thus $P \equiv Q$. However, if we choose a program $R = \{b \leftarrow \sim a\}$ as the context in which P and Q are placed, we notice that P and Q are not strongly equivalent, since $\mathbf{SM}(P \cup R) = \{\{a\}\} \neq \{\{a\}, \{b\}\} = \mathbf{SM}(Q \cup R)$. Thus $P \not\equiv_s Q$.

As discussed in the introduction, we concentrate on weak equivalence in this paper. In order to verify $P \equiv Q$, one needs to check that all stable models of P are also stable models of Q, and that all stable models of Q are stable models of P. In this way, the verification of $P \equiv Q$ involves two directions. In a naive approach, one explicitly enumerates all stable models of P and verifies them as stable models of Q, and vice versa. In the worst case, i.e. when the programs are equivalent, one has to find and check all stable models of both programs. This becomes easily infeasible if the number of stable models is high.

A counter-example for the equivalence of programs P and Q (in one direction) is an interpretation M such that $M \in \mathbf{SM}(P)$ and $M \notin \mathbf{SM}(Q)$. There are two possible reasons for M not being a stable model of Q: either $M \not\models Q_M$ ($\iff M \not\models Q$ [13]) or $M \models Q_M$ but M is not a minimal model of Q_M ($M \notin \mathbf{MM}(Q_M)$). Further on, we shall distinguish two types of counter-examples and present them as pairs. A counter-example of type T1 is a pair $\langle M, M\rangle$ such that $M \in \mathbf{SM}(P)$ and $M \not\models Q_M$; and a counter example of type T2 is a pair $\langle M, M'\rangle$ such that $M \in \mathbf{SM}(P)$, $M \models Q_M$, $M' \subset M$, and $M' \models Q_M$.

We write $\mathrm{CE}(P, Q)$ for the set of counter-examples of types T1 and T2 defined above. A counter-example can also be used to show that programs are not

strongly equivalent by forming an SE-model [29]. That is, in case of a type T1 counter-example, the pair $\langle M, M\rangle$ is an SE-model of P but not of Q and in case of type T2, $\langle M', M\rangle$ is an SE-model of Q but not of P. Thus $\mathrm{CE}(P,Q) \neq \emptyset$ implies $P \not\equiv_s Q$. It is also worth pointing out that in case of a counter-example $\langle M, M'\rangle$ of type T2, the model M' is not necessarily a minimal model of Q_M. The number of counter-examples of type T2 can be reduced by insisting on minimality, but then translations (to be presented in Section 4) need to be revised accordingly.

Next, we discuss the computational cost of equivalence testing. We assume the reader to be familiar with basic concepts of computational complexity [24] and introduce the languages corresponding to decision problems of our interest.

Definition 2. *For any finite DLPs P and Q, and $M \subseteq \mathrm{Hb}(P)$,*

- $P \in \mathrm{SM} \iff$ *there exists an interpretation* $M \subseteq \mathrm{Hb}(P)$ *s.t.* $M \in \mathbf{SM}(P)$,
- $\langle P, M\rangle \in \mathrm{NotMin} \iff$ *there exists an interpretation* $M' \subset M$ *s.t.* $M' \models P$,
- $\langle P, Q\rangle \in \mathrm{IMPR} \iff \mathbf{SM}(P) \subseteq \mathbf{SM}(Q)$,
- $\langle P, Q\rangle \in \mathrm{IMPL} \iff \langle Q, P\rangle \in \mathrm{IMPR}$, *and*
- $\langle P, Q\rangle \in \mathrm{EQV} \iff \mathbf{SM}(P) = \mathbf{SM}(Q) \iff P \equiv Q$.

The language SM for disjunctive logic programs is $\Sigma_2^{\mathbf{P}}$-complete [3], thus its complement $\overline{\mathrm{SM}}$ is $\Pi_2^{\mathbf{P}}$-complete. Restricted to normal logic programs, SM is only **NP**-complete [20]. The language NotMin is in **NP**, since we can construct a nondeterministic Turing machine deciding NotMin by selecting an interpretation $M' \subset M$ and checking whether $M' \models P$. Thus an **NP**-oracle can be used to decide whether there exists $M' \subset M$ such that $M' \models P$ for $M \subseteq \mathrm{Hb}(P)$.

Turner [29] shows that deciding the weak equivalence of DLPs is $\Pi_2^{\mathbf{P}}$-hard. It is indeed $\Pi_2^{\mathbf{P}}$-complete, since IMPR, IMPL, EQV are $\Pi_2^{\mathbf{P}}$-complete. The language IMPR is in $\Pi_2^{\mathbf{P}}$, since we can construct a nondeterministic **NP**-oracle Turing machine deciding $\overline{\mathrm{IMPR}}$ by first selecting an interpretation M and checking that $M \models P$, then using an oracle for NotMin to check that there exists no $M' \subset M$ s.t. $M' \models P_M$, thus $M \in \mathbf{SM}(P)$. If $M \not\models Q_M$, we accept $\langle P, Q\rangle$, as $\langle M, M\rangle$ is a counter-example of type T1. Otherwise, an oracle for NotMin is used to check whether there exists $M' \subset M$ s.t. $M' \models Q_M$. If $M \notin \mathbf{MM}(Q_M)$, we accept $\langle P, Q\rangle$, as there is a counter-example of type T2. IMPR is $\Pi_2^{\mathbf{P}}$-hard, since for an arbitrary DLP R, $R \in \overline{\mathrm{SM}} \iff R \notin \mathrm{SM} \iff \mathbf{SM}(R) = \emptyset$. Since $\mathbf{SM}(\{\bot\}) = \emptyset$, $\mathbf{SM}(R) \subseteq \mathbf{SM}(\{\bot\}) \iff \langle R, \{\bot\}\rangle \in \mathrm{IMPR}$. Similarly IMPL is $\Pi_2^{\mathbf{P}}$-complete and EQV is $\Pi_2^{\mathbf{P}}$-complete as $\mathrm{EQV} = \mathrm{IMPR} \cap \mathrm{IMPL}$.

Since EQV is $\Pi_2^{\mathbf{P}}$-complete, testing the weak equivalence of DLPs is as hard as deciding whether a DLP has stable models or not. Thus there is no complexity theoretical obstacle for developing a polynomial transformation for testing the equivalence of DLPs as introduced in [14] for the weight constraint programs. The question remains, however, whether it is possible to find a similar systematic way to obtain the transformation as in [14]. It is worth noting that deciding the equivalence of DLPs is strictly harder than deciding the equivalence of weight constraint programs[3], which forms only a **coNP**-complete decision problem.

[3] Unless the polynomial hierarchy collapses.

4 Translations for Equivalence Testing

In this section we present two translations for equivalence testing, a single translation and a two-phased one. We assume throughout this section that P and Q are two DLPs satisfying $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$. This is not a significant restriction, since any disjunctive logic program can be extended with rules of the form $a \leftarrow a$ without affecting the stable models of the program, as $\{a \leftarrow a\} \equiv_s \emptyset$.

The idea behind the first translation is to transform DLPs P and Q into one program $\mathrm{TR}(P, Q)$ that has a stable model iff P has a stable model that is not a stable model of Q. There are two possible reasons for $M \notin \mathbf{SM}(Q)$ corresponding to counter-examples of type T1 and T2: either $M \not\models Q_M$ (T1) or $M \models Q_M$, but $M \notin \mathbf{MM}(Q_M)$ (T2). The translation is used to capture these two possibilities. A further objective is that it should be easy to extract a counter-example for $P \equiv Q$ from a stable model N of the translation $\mathrm{TR}(P, Q)$.

For each atom $a \in \mathrm{Hb}(Q)$ we introduce new atoms $a^\bullet$ and a°. We define $A^\bullet = \{a^\bullet \mid a \in A\}$ and $A^\circ = \{a^\circ \mid a \in A\}$ for any set of atoms A. The Herbrand base of $\mathrm{TR}(P, Q)$ will be $\mathrm{Hb}(P) \cup \mathrm{Hb}(P)^\bullet \cup \mathrm{Hb}(P)^\circ \cup \{\mathrm{diff}, \mathrm{unsat}, \mathrm{unsat}^\bullet, \mathrm{ok}\}$, where $\mathrm{diff}, \mathrm{unsat}, \mathrm{unsat}^\bullet$ and ok are new atoms. Given $M \in \mathbf{SM}(P)$, the atoms in $\mathrm{Hb}(P)^\bullet$ are used to select a subset M' for M, $M' \subseteq M$. The intended meaning for the new atoms introduced in the translation $\mathrm{TR}(P, Q)$ is as follows.

unsat – indicates that $M \not\models Q_M$ (thus $M \not\models Q$ [13]).
$a^\bullet$ – denotes that atom $a \in M$ is in the sub-model M' searched for Q_M
a° – denotes that atom $a \in M$ is not in the sub-model M'.
$\mathrm{unsat}^\bullet$ – indicates that $M' \not\models Q_M$.
diff – indicates that $M' \neq M$, i.e. $M' \subset M$.
ok – indicates that a counter-example for the equivalence is found.

Definition 3. *Let* diff, unsat, $\mathrm{unsat}^\bullet$ *and* ok *be new atoms not appearing in* P *or in* Q. *The translation* $\mathrm{TR}(P, Q)$ *contains the following rules:*

1. *all the rules of* P *without modifications,*
2. *a rule* $\mathrm{unsat} \leftarrow B, {\sim}(A \cup C)$ *for each rule* $A \leftarrow B, {\sim}C \in Q$,
3. *rules* $a^\bullet \leftarrow a, {\sim}a^\circ, {\sim}\mathrm{unsat}$ *and* $a^\circ \leftarrow a, {\sim}a^\bullet, {\sim}\mathrm{unsat}$ *for each* $a \in \mathrm{Hb}(P)$,
4. *a rule* $\mathrm{unsat}^\bullet \leftarrow B^\bullet, {\sim}(A^\bullet \cup C), {\sim}\mathrm{unsat}$ *for each rule* $A \leftarrow B, {\sim}C \in Q$,
5. *a rule* $\mathrm{diff} \leftarrow a, {\sim}a^\bullet, {\sim}\mathrm{unsat}$ *for each atom* $a \in \mathrm{Hb}(P)$ *and*
6. *rules* $\mathrm{ok} \leftarrow \mathrm{unsat}$, $\mathrm{ok} \leftarrow \mathrm{diff}, {\sim}\mathrm{unsat}, {\sim}\mathrm{unsat}^\bullet$ *and* $\bot \leftarrow {\sim}\mathrm{ok}$.

Now, let us discuss the meaning of each individual part of the translation, the items below corresponding respectively to the items of Definition 3.

1. Capture a stable model M of P.
2. Check whether $M \models Q_M$. Thus, if $M \not\models Q_M$, then unsat is implied.
3. Select an interpretation M' s.t. $M' \subseteq M$. These rules force that if $a \in M$, then either a is in the sub-model candidate M' ($a^\bullet \in N$) or not ($a^\circ \in N$).
4. Check whether $(M')^\bullet \models (Q_M)^\bullet$, where $(M')^\bullet = N \cap \mathrm{Hb}(P)^\bullet$. Thus, if there exists a rule in Q_M that is not satisfied in M', unsat$^\bullet$ is implied.
5. Check that M' is a proper subset of M. That is, diff is implied if $M' \subset M$.
6. Summarize the reasons for M not being a stable model of Q. Either
   - T1: $M \not\models Q_M$ ($\Rightarrow$ unsat $\in N$), or
   - T2: $M \models Q_M$ ($\Rightarrow$ unsat $\notin N$) and $M \notin \mathbf{MM}(Q_M)$, since $M' \subset M$ ($\Rightarrow$ diff $\in N$) and $M' \models Q_M$ ($\Rightarrow$ unsat$^\bullet \notin N$).

   A stable model for the translation exists iff such a reason exists, thus the integrity constraint ensures that every stable model contains the atom ok.

A counter-example can be easily constructed from a stable model N of $\mathrm{TR}(P, Q)$. Clearly ok $\in N$. If unsat $\in N$, then $M = N \cap \mathrm{Hb}(P) \in \mathbf{SM}(P)$ and $M \not\models Q_M$. Thus $\langle M, M\rangle$ is a counter-example of type T1. On the other hand, if unsat $\notin N$, then also diff $\in N$ and unsat$^\bullet \notin N$. Now, $M = N \cap \mathrm{Hb}(P) \in \mathbf{SM}(P)$ and $M \notin \mathbf{MM}(Q_M)$. Furthermore, $M' = \{a \mid a^\bullet \in N \cap \mathrm{Hb}(P)^\bullet\} \subset M$ and $M' \models Q_M$. Thus $\langle M, M'\rangle$ is a counter-example of type T2.

Example 2. Consider DLPs $P = \{a \mid b\}$ and $Q = \{a \leftarrow {\sim}b\}$. Now, $\mathbf{SM}(P) = \{\{a\}, \{b\}\}$ and $\mathbf{SM}(Q) = \{\{a\}\}$. The translation $\mathrm{TR}(P, Q)$ is

$$\{\ a \mid b.\ \ \mathrm{unsat} \leftarrow {\sim}a, {\sim}b.\ \ a^\bullet \leftarrow a, {\sim}a^\circ, {\sim}\mathrm{unsat}.\ \ a^\circ \leftarrow a, {\sim}a^\bullet, {\sim}\mathrm{unsat}.$$
$$b^\bullet \leftarrow b, {\sim}b^\circ, {\sim}\mathrm{unsat}.\ \ b^\circ \leftarrow b, {\sim}b^\bullet, {\sim}\mathrm{unsat}.\ \ \mathrm{unsat}^\bullet \leftarrow {\sim}a^\bullet, {\sim}b, {\sim}\mathrm{unsat}.$$
$$\mathrm{diff} \leftarrow a, {\sim}a^\bullet, {\sim}\mathrm{unsat}.\ \ \mathrm{diff} \leftarrow b, {\sim}b^\bullet, {\sim}\mathrm{unsat}.\ \ \mathrm{ok} \leftarrow \mathrm{unsat}.$$
$$\mathrm{ok} \leftarrow \mathrm{diff}, {\sim}\mathrm{unsat}, {\sim}\mathrm{unsat}^\bullet.\ \ \bot \leftarrow {\sim}\mathrm{ok}\ \}.$$

Consider a model candidate $N = \{b, b^\circ, \mathrm{diff}, \mathrm{ok}\}$. The reduct $\mathrm{TR}(P, Q)_N$ is $\{a \mid b.\ \ a^\bullet \leftarrow a.\ \ a^\circ \leftarrow a.\ \ b^\circ \leftarrow b.\ \ \mathrm{diff} \leftarrow a.\ \ \mathrm{diff} \leftarrow b.\ \ \mathrm{ok} \leftarrow \mathrm{unsat}.\ \ \mathrm{ok} \leftarrow \mathrm{diff}.\}$. $N \in \mathbf{SM}(\mathrm{TR}(P, Q))$, since $\mathbf{MM}(\mathrm{TR}(P, Q)_N) = \{\{a, a^\bullet, a^\circ, \mathrm{diff}, \mathrm{ok}\}, \{b, b^\circ, \mathrm{diff}, \mathrm{ok}\}\}$. Since the translation has a stable model, we can conclude, that $P \not\equiv Q$, and the counter-example is $\langle M, M'\rangle$ where $M = N \cap \mathrm{Hb}(P) = \{b\}$ and $M' = \{a \mid a^\bullet \in N \cap \mathrm{Hb}(P)^\bullet\} = \emptyset$. Now $M \in \mathbf{SM}(P)$ and $M \notin \mathbf{MM}(Q_M) = \{\emptyset\}$. Also, $M' \subset M$ and $M' \models Q_M$, since $Q_M = \emptyset$. Since unsat $\notin N$, the counter-example is of type T2. As both stable models of P are models of Q, there is no counter-example of type T1. This can be also verified from the translation: the first two rules in $\mathrm{TR}(P, Q)$ ensure that unsat cannot belong to any stable model of $\mathrm{TR}(P, Q)$.

Finding a counter-example for equivalence divides clearly in two cases (types T1 and T2) which suggests to perform testing in two phases. The idea is as follows. In the first phase, we use a translation $\mathrm{TR}_1(P, Q)$ to test whether all the stable models of P are *models* of Q_M. If there exists $M \in \mathbf{SM}(P)$ such that $M \not\models Q_M$, then $\mathrm{TR}_1(P, Q)$ has a stable model and we have found a counter-example of type T1. Otherwise, we will continue to the second phase, where the second translation $\mathrm{TR}_2(P, Q)$ is used to verify whether every $M \in \mathbf{SM}(P)$

is a minimal model of Q_M. The required translations can be obtained rather easily by simplifying $\mathrm{TR}(P,Q)$. The two-phased approach can be motivated by computational arguments. Counter-examples of type T1 can be found using a compact translation, whereas the translation for finding a counter-example of type T2 is more involved. Thus counter-examples of type T2 should be of interest only if counter-examples of type T1 do not exist. In this way, the search space is divided (conditionally) in two parts. Moreover, the translation of the type T2 counter-examples can be simplified, since it is known that every $M \in \mathbf{SM}(P)$ is necessarily a model of Q_M. In the definitions of the translations $\mathrm{TR}_1(P,Q)$ and $\mathrm{TR}_2(P,Q)$ below, we use the same notations and atoms as in Definition 3.

Definition 4. *Let* unsat *be a new atom not appearing in P or in Q. The translation $\mathrm{TR}_1(P,Q)$ contains the rules from items 1 and 2 from Definition 3 and in addition a rule* $\bot \leftarrow {\sim}\mathrm{unsat}$.

We can easily construct a counter-example of type T1 for $P \equiv Q$ from a stable model of the translation $\mathrm{TR}_1(P,Q)$. If $N \in \mathbf{SM}(\mathrm{TR}_1(P,Q))$, then $M \in \mathbf{SM}(P)$ and $M \not\models Q$ ($\Rightarrow M \not\models Q_M$), where $M = N \cap \mathrm{Hb}(P)$. Thus $\langle M, M\rangle$ is a counter-example of type T1. If $\mathbf{SM}(\mathrm{TR}_1(P,Q)) = \emptyset$ is the case, then we have to perform further testing using another translation $\mathrm{TR}_2(P,Q)$ given below.

Definition 5. *Let* diff *and* $\mathrm{unsat}^\bullet$ *be new atoms not appearing in P or in Q. The translation $\mathrm{TR}_2(P,Q)$ contains the following rules:*

1. *all the rules of P without modifications,*
2. *rules $a^\bullet \leftarrow a, {\sim}a^\circ$ and $a^\circ \leftarrow a, {\sim}a^\bullet$ for each atom $a \in \mathrm{Hb}(P)$,*
3. *a rule* $\mathrm{unsat}^\bullet \leftarrow B^\bullet, {\sim}(A^\bullet \cup C)$ *for each rule $A \leftarrow B, {\sim}C \in Q$,*
4. *a rule* $\mathrm{diff} \leftarrow a, {\sim}a^\bullet$ *for each atom $a \in \mathrm{Hb}(P)$ and*
5. *rules* $\bot \leftarrow {\sim}\mathrm{diff}$ *and* $\bot \leftarrow \mathrm{unsat}^\bullet$.

Stable models of the translation $\mathrm{TR}_2(P,Q)$ yield counter-examples as follows. Since $\mathbf{SM}(\mathrm{TR}_1(P,Q)) = \emptyset$, there exists no counter-example of type T1, i.e. if $M \in \mathbf{SM}(P)$, then $M \models Q_M$. If $N \in \mathbf{SM}(\mathrm{TR}_2(P,Q))$, then $M = N \cap \mathrm{Hb}(P) \in \mathbf{SM}(P)$, $M \models Q_M$, and $M \notin \mathbf{MM}(Q_M)$. Also, $M' = \{a \mid a^\bullet \in N \cap \mathrm{Hb}(P)^\bullet\} \subset M$ and $M' \models Q_M$. Thus $\langle M, M'\rangle$ is a counter-example of type T2.

In fact, there is a tight correspondence between the set of counter-examples $\mathrm{CE}(P,Q)$ and $\mathbf{SM}(\mathrm{TR}(P,Q))$ to be made explicit as follows. First we define a mapping from $\mathrm{CE}(P,Q)$ to $\mathbf{SM}(\mathrm{TR}(P,Q))$. Given a pair of interpretations $\langle M, M'\rangle$ of P such that $M' \subseteq M$, we define $\mathrm{EXT}_{P,Q}(M,M') = M \cup \{\mathrm{unsat}, \mathrm{ok}\}$, if $M = M'$, and $\mathrm{EXT}_{P,Q}(M,M') = M \cup \{\mathrm{diff}, \mathrm{ok}\} \cup \{a^\bullet \mid a \in M'\} \cup \{a^\circ \mid a \in M \setminus M'\}$, otherwise. The other direction follows. Given an interpretation of the translation $\mathrm{TR}(P,Q)$, let $\mathrm{PROJ}_{P,Q}(N) = \langle N \cap \mathrm{Hb}(P), N \cap \mathrm{Hb}(P)\rangle$, if $\mathrm{unsat} \in N$, and $\mathrm{PROJ}_{P,Q}(N) = \langle N \cap \mathrm{Hb}(P), \{a \in \mathrm{Hb}(P) \mid a^\bullet \in N\}\rangle$, otherwise.

Theorem 1. *If $\langle M, M'\rangle \in \mathrm{CE}(P,Q)$, then the set $N = \mathrm{EXT}_{P,Q}(M,M') \in \mathbf{SM}(\mathrm{TR}(P,Q))$ such that $\mathrm{PROJ}_{P,Q}(N) = \langle M, M'\rangle$. If $N \in \mathbf{SM}(\mathrm{TR}(P,Q))$, then the pair $\langle M, M'\rangle = \mathrm{PROJ}_{P,Q}(N) \in \mathrm{CE}(P,Q)$ such that $N = \mathrm{EXT}_{P,Q}(M,M')$.*

As a consequence of Theorem 1, the mappings $\mathrm{EXT}_{P,Q}$ and $\mathrm{PROJ}_{P,Q}$ are bijections implying that the counter-examples in $\mathrm{CE}(P,Q)$ and the stable models of $\mathrm{TR}(P,Q)$ are in a one-to-one correspondence. This relationship implies the correctness of our method for verifying the equivalence of DLPs.

Corollary 1. *For DLPs P and Q such that* $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$, $P \equiv Q \iff \mathbf{SM}(\mathrm{TR}(P,Q)) = \emptyset$ *and* $\mathbf{SM}(\mathrm{TR}(Q,P)) = \emptyset$.

Let us then turn our attention to the correctness of the two-phased translation $[\mathrm{TR}_1(P,Q), \mathrm{TR}_2(P,Q)]$. In this case, we partition the set of counter-examples $\mathrm{CE}(P,Q)$ into $\mathrm{CE}_1(P,Q) \cup \mathrm{CE}_2(P,Q)$ by using the type of counter-examples as a criterion. The mappings $\mathrm{EXT}_{P,Q}$ and $\mathrm{PROJ}_{P,Q}$ are revised accordingly by dropping the atom "ok" not appearing in $[\mathrm{TR}_1(P,Q), \mathrm{TR}_2(P,Q)]$. The correctness of the translation is established phase-wise.

Theorem 2. *If* $\langle M, M\rangle \in \mathrm{CE}_1(P,Q)$, *then there is* $N = \mathrm{EXT}_{P,Q}(M,M) \in \mathbf{SM}(\mathrm{TR}_1(P,Q))$ *such that* $\mathrm{PROJ}_{P,Q}(N) = \langle M, M\rangle$. *If* $N \in \mathbf{SM}(\mathrm{TR}_1(P,Q))$, *then* $\langle M, M\rangle = \mathrm{PROJ}_{P,Q}(N) \in \mathrm{CE}_1(P,Q)$ *such that* $N = \mathrm{EXT}_{P,Q}(M,M)$.

Theorem 3. *Suppose that* $\mathrm{CE}_1(P,Q) = \emptyset$, *or equivalently* $\mathbf{SM}(\mathrm{TR}_1(P,Q)) = \emptyset$. *If* $\langle M, M'\rangle \in \mathrm{CE}_2(P,Q)$, *then there is* $N = \mathrm{EXT}_{P,Q}(M,M') \in \mathbf{SM}(\mathrm{TR}_2(P,Q))$ *such that* $\mathrm{PROJ}_{P,Q}(N) = \langle M, M'\rangle$. *If* $N \in \mathbf{SM}(\mathrm{TR}_2(P,Q))$, *then* $\langle M, M'\rangle = \mathrm{PROJ}_{P,Q}(N) \in \mathrm{CE}_2(P,Q)$ *such that* $N = \mathrm{EXT}_{P,Q}(M,M')$.

By a slight notational generalization, we define the set of stable models $\mathbf{SM}([\mathrm{TR}_1(P,Q), \mathrm{TR}_2(P,Q)])$ as $\mathbf{SM}(\mathrm{TR}_1(P,Q))$, if $\mathbf{SM}(\mathrm{TR}_1(P,Q)) \neq \emptyset$, and as $\mathbf{SM}(\mathrm{TR}_2(P,Q))$, otherwise. Then the correctness of the two-phased translation-based method can be formulated in analogy to Corollary 1.

Corollary 2. *For DLPs P and Q such that* $\mathrm{Hb}(P) = \mathrm{Hb}(Q)$, $P \equiv Q \iff \mathbf{SM}([\mathrm{TR}_1(P,Q), \mathrm{TR}_2(P,Q)]) = \emptyset$ *and* $\mathbf{SM}([\mathrm{TR}_1(Q,P), \mathrm{TR}_2(Q,P)]) = \emptyset$.

5 Experiments

Translation functions TR and $[\mathrm{TR}_1, \mathrm{TR}_2]$ presented in Section 4 have been implemented in C under Linux. The translator DLPEQ [23] takes two DLPs as input and produces by default the translation TR as its output. Translations TR_1 and TR_2 can be produced using command line options. The program assumes the internal format of GNT and thus the front-end LPARSE can be used. Also, the translation is printed in a textual form upon request. It is important to note that DLPEQ checks that the *visible* Herbrand bases of the DLPs being compared are the same (a visible atom has a name in the symbol table of the DLP). The front-end may produce some *invisible* atoms that are neglected in the comparison. To support programs produced by LPARSE, such atoms keep their roles in the respective programs. Also, DLPEQ even enables equivalence testing of DLPs P and Q such that $\mathrm{Hb}(P) \neq \mathrm{Hb}(Q)$: the *visible* Herbrand bases of P and Q are extended to be the same by adding (useless) rules of the form $a \leftarrow a$.

To assess the feasibility of DLPEQ in practice we ran tests to compare running times of the DLPEQ and DLPEQ-2 (two-phased translation) approach with a NAIVE approach, in which one (i) generates a stable model M of P, (ii) tests whether M is a stable model of Q, (iii) stops if not, and otherwise continues from step (i) until all stable models of P get enumerated. A similar testing has to be carried out in the other direction to establish $P \equiv Q$. There is still room for optimization in all the approaches. If one finds a counter-example in one direction, then $P \not\equiv Q$ holds immediately and there is no need to test the other direction except to perform a thorough analysis. We decided not to use this optimization, since running times turned out to scale differently depending on the direction. Thus we count running times in both directions.

In all the approaches the GNT system is responsible for the computation of stable models. In the DLPEQ approach, the total running time (in one direction) is the running time needed for trying to compute *one* stable model of the translation produced by DLPEQ. In the DLPEQ-2 approach, the total running time (in one direction) is the time needed for trying to compute *one* stable model of the translation TR_1 plus the time needed for trying to compute *one* stable model of the translation TR_2, if there exists no stable models for TR_1. In both approaches the actual translation times are not taken into account, as they are negligible. In the NAIVE approach, we try to exclude possible overhead due to implementing the NAIVE approach as a shell script. Thus the total running time (in one direction) consists only of the running time for finding the necessary (but not necessarily all) stable models of P plus the individual running times for testing that the stable models found are also stable models of Q. These tests are realized in practice by adding M as a compute statement to Q. The tests were run under the Debian GNU/Linux 2.4.18 operating system on a 450MHz Intel Pentium III computer with 256MB of memory.

We use a $\Sigma_2^{\mathbf{P}}$-complete problem of finding a minimal model of a set of clauses containing specified atoms [3] as our first test problem. Each clause $a_1 \vee \cdots \vee a_n \vee \neg b_1 \vee \cdots \vee \neg b_m$ of an instance is translated into a rule $a_1 | \ldots | a_n \leftarrow b_1, \ldots, b_m$ and a rule $\bot \leftarrow \sim c_i$ is included for each specified atom c_i. The resulting DLP has a stable model iff there is a minimal model of the clauses containing all the specified atoms c_i. We generate DLPs that solve an instance of a random 3-sat problem and add the extra rules for random atoms c_i, for $i = 1, \ldots, \lfloor 2v/100 \rfloor$, where v is the number of atoms in the instance. First we keep the clauses-to-variables ratio c/v constant at 3.5. The DPLs generated from such instances typically have several stable models. To simulate a sloppy programmer making mistakes, we drop *one* random rule from each program. Then we test the equivalence of the modified (Drop-1) and the original DLP to see if making a mistake affects the stable models of the program or not. Thus, the tested pairs of DLPs include both equivalent and nonequivalent cases. We vary the number of variables v from 50 to 100 with steps of 5. For each number of variables we repeat the test 100 times – generating each time a new random instance. For the problem sizes used, the percentage of nonequivalent cases varies between 60 and 90. The maximum, average and minimum running times of each approach are plotted in

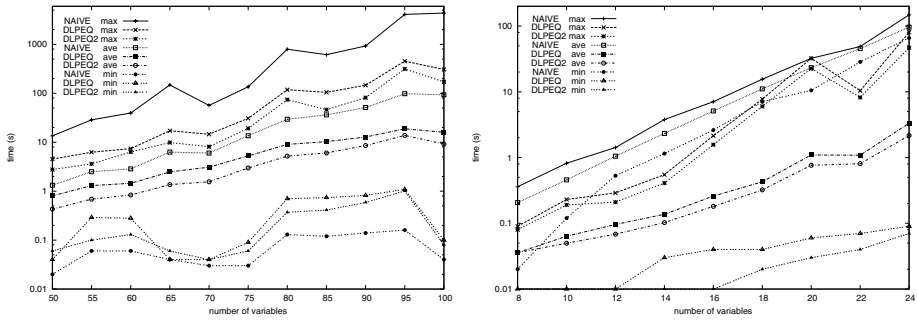


Fig. 1. Equivalence testing of random 3-sat instances with the ratio $c/v = 3.5$ (left) and random 2-qbf instances with the ratio $c/v = 3.5$ (right).

Fig. 1 (left). The DLPEQ approach turns out to be faster than the NAIVE one and the difference in running times increases as instances grow in size. Furthermore, the DLPEQ-2 approach outperforms the DLPEQ approach.

Next we generate programs as in the previous experiment, but keep the number of variables constant at $v = 100$. We vary the ratio c/v from 3.25 to 4.50 with steps of 0.25 to see how the translation-based approaches perform compared to the NAIVE one as the tested DLPs change from ones having many stable models to programs having no stable models. With lower values of c/v the DLPEQ and DLPEQ-2 approaches are clearly superior to the NAIVE one. As the ratio increases the performance of the NAIVE approach gradually improves and finally outperforms the two other approaches.

Our second test problem is a $\Sigma_2^{\mathbf{P}}$-complete problem of deciding the validity of a 2-qbf (quantified Boolean formula). A 2-qbf instance is of the form $\Phi = \exists X \forall Y \phi$, where $X \cap Y = \emptyset$ and ϕ is a 3-sat formula in DNF over the set of variables $X \cup Y$. In our experiment $|X| = |Y|$ and $v = |X \cup Y|$. We generate random 2-qbf instances according to model A presented by Gent and Walsh [8], i.e. each clause in ϕ contains at least two variables that are universally quantified. We transform a generated random instance Φ to a DLP using a transformation presented by Eiter and Gottlob [3]. We test the equivalence of the program and its modified version (Drop-1). We keep c/v constant at 3.5 and vary the number of variables v from 10 to 24 with steps of 2. For each v we repeat the test 100 times, generating each time a new random instance. For the problem sizes used, the percentage of nonequivalent cases is approximately 50–60. The maximum, average and minimum running times of all the approaches are plotted in Fig. 1 (right). The DLPEQ approach is clearly faster than the NAIVE one. The difference in running times increases as instances grow in size. Furthermore, the DLPEQ-2 approach performs slightly better that the DLPEQ approach.

6 Conclusions

This paper extends the translation-based approach for testing the equivalence of logic programs under the stable model semantics [14] to the case of disjunctive

programs. Two systematic transformations are presented in order to reduce the problem of testing whether two logic programs are equivalent to the problem of computing stable models for a disjunctive program. As a consequence, existing search engines such as DLV and GNT can be used for the search of counter-examples, and there is no need to develop a special purpose system for this task. Our implementation, a translator program called DLPEQ, is compatible with GNT and it enables the equivalence testing of disjunctive programs in practice.

The preliminary experiments reported in Section 5 suggest that the translation-based approach is superior to a naive cross-checking approach when the programs possess many stable models, although the length of the translation exceeds the sum of the lengths of the programs being tested. To our understanding, this is because the translation provides an explicit specification of a counter-example for $P \equiv Q$ which enables GNT to prune search space. This is not possible in the NAIVE approach where the stable models of P and Q are explicitly enumerated. However, if the programs under comparison have few/no stable models, then the naive approach is likely to be faster. A further observation is that the two-phased translation $[\mathrm{TR}_1(P,Q), \mathrm{TR}_2(P,Q)]$ is more efficient than the approach based on a single translation $\mathrm{TR}(P,Q)$. Thus it seems a good idea to compute the two types of counter-examples in isolation.

Finally, let us sketch future work. It seems that even more general classes of logic programs can be covered via suitable translations. For instance, the weak equivalence of *nested programs* [16] can be addressed by implementing the transformation of nested programs into disjunctive programs [25]. On the other hand, determining the strong equivalence of nested (as well as disjunctive) programs is computationally less complex, which suggests using an NP solver for the task. E.g., Lin [18] gives a transformation that enables the use of SAT solvers for testing strong equivalence. Moreover, DLPEQ should be extended to cover other equivalence relations such as uniform [2] and strong equivalence [16].

References

1. M. Balduccini, M. Gelfond, R. Watson, and M. Nogueira. The USA-advisor: A case study in answer set planning. In *Proc. of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001)*, pages 439–442, Vienna, Austria, September 2001. Springer. LNAI 2173.
2. T. Eiter and M. Fink. Uniform equivalence of logic programs under the stable model semantics. In *Proc. of the 19th International Conference on Logic Programming (ICLP'03)*, 2003. To appear.
3. T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Math. and AI*, 15:289–323, 1995.
4. P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. Available at `http://www.cs.utexas.edu/users/vl/papers.html`, 2003. Unpublished draft.
5. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. MIT Press.

6. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proc. of the 7th International Conference on Logic Programming*, pages 579–597, Jerusalem, Israel, June 1990. MIT Press.
7. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
8. I. Gent and T. Walsh. Beyond NP: the QSAT phase transition. In *Proc. of the 16th National Conference on Artificial Intelligence and the 11th Conference on Innovative Applications of Artificial Intelligence*, pages 648–653, Orlando, Florida, USA, July 1999. AAAI, MIT Press.
9. K. Inoue and C. Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35(1):39–78, 1998.
10. T. Janhunen. On the effect of default negation on the expressiveness of disjunctive rules. In *Logic Programming and Nonmonotonic Reasoning, Proc. of the 6th International Conference*, pages 93–106, Vienna, Austria, September 2001. Springer.
11. T. Janhunen. LPEQ 1.13 – a tool for testing the equivalence of logic programs. `http://www.tcs.hut.fi/Software/lpeq/`, 2002. Computer Program.
12. T. Janhunen. GnT 2 – a tool for computing disjunctive stable models. `http://www.tcs.hut.fi/Software/gnt/`, 2003. Computer Program.
13. T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. CoRR: cs.AI/0303009, March 2003.
14. T. Janhunen and E. Oikarinen. Testing the equivalence of logic programs under stable model semantics. In *Logics in Artificial Intelligence, Proc. of the 8th European Conference*, pages 493–504, Cosenza, Italy, September 2002. Springer.
15. N. Leone et al. The DLV system for knowledge representation and reasoning. CoRR: cs.AI/0211004, September 2003.
16. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
17. V. Lifschitz, L.R. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Math. and AI*, 25(3–4):369–389, 1999.
18. F. Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In D. Fensel et al., editors, *Principles of Knowledge Representation and Reasoning: Proc. of the 8th International Conference*, pages 170–176, Tolouse, France, April 2002. Morgan Kaufmann.
19. J.W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1987.
20. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.
21. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, 1999.
22. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Math. and AI*, 25(3–4):241–273, 1999.
23. E. Oikarinen. DLPEQ 1.9 – a tool for testing the equivalence of disjunctive logic programs. `http://www.tcs.hut.fi/Software/lpeq/`, 2003. Computer Program.
24. C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
25. D. Pearce, V. Sarsakov, T. Schaub, H. Tompits, and S. Woltran. A polynomial translation of logic programs with nested expressions into disjunctive logic programs: Preliminary report. In *Proc. of the 18th International Conference on Logic Programming*, pages 405–420, Copenhagen, Denmark, July–August 2002. Springer.
26. D. Pearce, H. Tompits, and S. Woltran. Encodings for equilibirium logic and logic programs with nested expressions. In *Proc. of the 10th Portuguese Conference on Artificial Intelligence*, pages 306–320, Porto, Portugal, December 2001. Springer.

27. T. Przymusinski. Extended stable semantics for normal and disjunctive logic programs. In *Proc. of the 7th International Conference on Logic Programming*, pages 459–477. MIT Press, 1990.
28. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
29. H. Turner. Strong equivalence made easy: Nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4–5):609–622, 2003.

Uniform Equivalence for Equilibrium Logic and Logic Programs

David Pearce[1]* and Agustín Valverde[2]**

[1] Universidad Rey Juan Carlos (Madrid, Spain)
d.pearce@escet.urjc.es
[2] Universidad de Málaga (Málaga, Spain)
a_valverde@ctima.uma.es

Abstract. For a given semantics, two logic programs Π_1 and Π_2 can be said to be *equivalent* if they have the same intended models and *strongly equivalent* if for any program X, $\Pi_1 \cup X$ and $\Pi_2 \cup X$ are equivalent. Eiter and Fink have recently studied and characterised under answer set semantics a further, related property of *uniform* equivalence, where the extension X is required to be a set of atoms. We extend their main results to propositional theories in equilibrium logic and describe a tableaux proof system for checking the property of uniform equivalence. We also show that no new forms of equivalence are obtained by varying the logical form of expressions in the extension X. Finally, some examples are studied including special cases of nested and generalized rules.

1 Introduction

Concepts of program equivalence are important in both the theory and practice of logic programming. In terms of theory, knowing in general terms when logic programs are equivalent provides important information about their mathematical properties. In practical terms, knowing that two programs are equivalent may mean in certain contexts that one can be replaced by the other without loss. In answer set programming (ASP), the property of having the same answer sets can be viewed as the simplest kind of equivalence. Two programs with this property respond to queries in the same way: they have the same credulous and the same skeptical consequences. But they need not be inter-substitutable without loss in all contexts. To guarantee this property in the most general case, a notion of *strong* equivalence is needed. Two programs Π_1 and Π_2 are said to be *strongly equivalent* iff for any program X, $\Pi_1 \cup X$ and $\Pi_2 \cup X$ are equivalent, ie have the same answer sets. Strong equivalence in ASP is a powerful property that turns out to be easier to verify than ordinary equivalence. Lifschitz, Pearce and Valverde [13] showed that in answer set semantics programs are strongly equivalent if and only if they are equivalent in a certain non-classical logic called here-and-there (with strong negation), which we denote here by $\mathbf{N}_5$.[1] In the case of ordinary equivalence one has

* Partially supported by CICyT project TIC-2003-9001-C02
** Partially supported by CICyT project TIC-2003-9001-C01 and Junta de Andalucía project TIC-115.

[1] $\mathbf{N}_5$ is also a maximal logic with this property. Other logics capturing strong equivalence are described in [10].

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 194–206, 2004.

the harder task to verify that Π_1 and Π_2 have the same minimal $\mathbf{N}_5$ models of a special type, called equilibrium models, introduced in [14]. They correspond to answer sets.

Besides strong equivalence one may consider weaker concepts that still permit one program to be substituted for another in certain well-defined settings. One such notion is that of *uniform* or u-equivalence, defined as above but restricted to the case where X is a set of atoms. This concept is of interest when one is dealing with a fixed set of rules, or intensional knowledge base, and a varying set of facts or extensional knowledge component. It may also be relevant in other applications of ASP, eg in planning and diagnosis where there is a fixed background theory, and plans (resp. diagnostic explanations) are sequences (resp. sets) of atomic propositions. Uniform equivalent background theories will generate equivalent plans (resp. explanations).

The u-equivalence of logic programs under answer set semantics has recently been studied by Eiter and Fink [3]. For finite programs they show that u-equivalence can be neatly characterised in terms of certain maximal models. A weaker semantic property is demonstrated for the infinite case where such maximal models are not guaranteed to exist. They also look at several special classes of programs, such as Horn and head-cycle free programs, and provide complexity results for the general and several special cases. Here we extend the work of Eiter and Fink in several directions. First, their characterisations of uniform equivalence are proved for disjunctive programs using a notion of SE-model, introduced in [16]. They observe that this is essentially equivalent to the models of here-and-there logic and that the results generalise to programs with strong negation and even nested expressions. We shall prove the main characterisation results directly for theories in equilibrium logic using here-and-there models. This simplifies the proofs, generalises the results to the full propositional language and yields by the well-known properties of equilibrium logic the corresponding results for programs with strong negation and nested expressions without further ado. In the case of the characterisation applicable to infinite theories (Theorem 4 below), we strengthen the results slightly by simplifying part of the sufficiency condition. Secondly, we consider the question whether in the definition of uniform equivalence placing other restrictions on the logical form of sentences in the extension X yields new forms of equivalence lying 'between' uniform and strong equivalence. The answer is no. If rules involving implication are permitted in X, then strong equivalence is the appropriate concept. Otherwise, for implication-free formulas of any other logical type, the equivalence in question is equivalent to uniform. Thirdly, we consider a proof system for checking the property of uniform equivalence. For this, since we express equivalence using ordinary logical models in $\mathbf{N}_5$, we can adapt a tableau proof system for $\mathbf{N}_5$ that was studied in an earlier paper [15].

In [3] several examples are given of logic programs that are uniform but not strongly equivalent. A feature of equilibrium logic is that it allows one to represent programs with *conditional* rules, ie expressions of the form $\alpha \rightarrow (\beta \rightarrow \gamma)$ or $(\alpha \rightarrow \beta) \rightarrow \gamma$. It is interesting to consider when such rules are equivalent to ordinary programs with nested expressions, eg when can one replace $(p \rightarrow q) \rightarrow r$ by $(\neg p \vee q) \rightarrow r$. In general the last two expressions are u-equivalent but not strongly equivalent. In Sections 4 and 5 we consider some cases of this kind.

2 Equilibrium Logic

We work throughout in the nonclassical logic of here-and-there with strong negation $\mathbf{N}_5$ and its nonmonotonic extension, equilibrium logic [14], which generalises answer set semantics for logic programs to arbitrary propositional theories, see eg [13]. We give only a very brief overview of equilibrium logic here. For more details the reader is referred to [14,13,15] and the logic texts cited below.

Formulas of $\mathbf{N}_5$ are built-up in the usual way using the logical constants: $\wedge$, $\vee$, $\rightarrow$, $\neg$, $\sim$, standing respectively for conjunction, disjunction, implication, weak (or intuitionistic) negation and strong negation. The axioms and rules of inference for $\mathbf{N}_5$ are those of intuitionistic logic (see eg [17]) together with:

1. the axiom schema $(\neg\alpha \rightarrow \beta) \rightarrow (((\beta \rightarrow \alpha) \rightarrow \beta) \rightarrow \beta)$, which characterises the 3-valued here-and-there logic of Heyting [9], and Gödel [5] (hence it is sometimes known as Gödel's 3-valued logic).
2. the following axiom schemata involving strong negation taken from the calculus of Vorob'ev [18,19] (where '$\alpha \leftrightarrow \beta$' abbreviates $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$):

N1. $\sim(\alpha \rightarrow \beta) \leftrightarrow \alpha \wedge \sim\beta$	**N2.** $\sim(\alpha \wedge \beta) \leftrightarrow \sim\alpha \vee \sim\beta$
N3. $\sim(\alpha \vee \beta) \leftrightarrow \sim\alpha \wedge \sim\beta$	**N4.** $\sim\sim\alpha \leftrightarrow \alpha$
N5. $\sim\neg\alpha \leftrightarrow \alpha$	**N6.** (for atomic α) $\quad \sim\alpha \rightarrow \neg\alpha$

The model theory of $\mathbf{N}_5$ is based on the usual Kripke semantics for Nelson's constructive logic $\mathbf{N}$ (see eg. [7,17]), but $\mathbf{N}_5$ is complete for Kripke frames $\mathcal{F} = \langle W, \leq\rangle$ (where as usual W is the set of point or worlds and $\leq$ is a partial-ordering on W) having exactly two worlds say h ('here') and t ('there') with $h \leq t$. As usual a *model* is a frame together with an assignment i that associates to each element of W a set of *literals*,[2] such that if $w \leq w'$ then $i(w) \subseteq i(w')$. An assignment is then extended inductively to all formulas via the usual rules for conjunction, disjunction, implication and (weak) negation in intuitionistic logic together with the following rules governing strongly negated formulas:

$$
\begin{array}{rlrcl}
\sim(\varphi \wedge \psi) \in i(w) & \text{iff} & \sim\varphi \in i(w) & \text{or} & \sim\psi \in i(w) \\
\sim(\varphi \vee \psi) \in i(w) & \text{iff} & \sim\varphi \in i(w) & \text{and} & \sim\psi \in i(w) \\
\sim(\varphi \rightarrow \psi) \in i(w) & \text{iff} & \varphi \in i(w) & \text{and} & \sim\psi \in i(w) \\
\sim\neg\varphi \in i(w) & \text{iff} & \varphi \in i(w) & & \\
\sim\sim\varphi \in i(w) & \text{iff} & \varphi \in i(w) & &
\end{array}
$$

It is convenient to represent an $\mathbf{N}_5$-model as an ordered pair $\langle H, T\rangle$ of sets of literals, where $H = i(h)$ and $T = i(t)$ under a suitable assignment i. By $h \leq t$, it follows that $H \subseteq T$. Again, by extending i inductively we know what it means for an arbitrary formula φ to be true in a model $\langle H, T\rangle$.

A formula φ is true in a here-and-there model $\mathcal{M} = \langle H, T\rangle$ in symbols $\mathcal{M} \models \varphi$, if it is true at each world in $\mathcal{M}$. A formula φ is said to be *valid* in $\mathbf{N}_5$, in symbols $\models \varphi$, if it is true in all here-and-there models. Logical consequence for $\mathbf{N}_5$ is understood as follows: φ is said to be an $\mathbf{N}_5$-consequence of a set Π of formulas, written $\Pi \models \varphi$, iff

[2] We use the term 'literal' to denote an atom, or atom prefixed by strong negation.

for all models $\mathcal{M}$ and any world $w \in \mathcal{M}$, $\mathcal{M}, w \models \Pi$ implies $\mathcal{M}, w \models \varphi$. Equivalently this can be expressed by saying that φ is true in all models of Π. Further properties of $\mathbf{N}_5$ are studied in [11].

Equilibrium models are special kinds of minimal $\mathbf{N}_5$ Kripke models. We first define a partial ordering $\trianglelefteq$ on $\mathbf{N}_5$ models that will be used both to characterise the equilibrium property as well as the property of uniform equivalence.

Definition 1. *Given any two models $\langle H, T\rangle$, $\langle H', T'\rangle$, we set $\langle H, T\rangle \trianglelefteq \langle H', T'\rangle$ if $T = T'$ and $H \subseteq H'$.*

Definition 2. *Let Π be a set of $\mathbf{N}_5$ formulas and $\langle H, T\rangle$ a model of Π.*

1. *$\langle H, T\rangle$ is said to be* total *if $H = T$.*
2. *$\langle H, T\rangle$ is said to be an* equilibrium *model if it is minimal under $\trianglelefteq$ among models of Π, and it is total.*

In other words a model $\langle H, T\rangle$ of Π is in equilibrium if it is total and there is no model $\langle H', T\rangle$ of Π with $H' \subset H$. Equilibrium logic is the logic determined by the equilibrium models of a theory. It generalises answer set semantics in the following sense. For all the usual classes of logic programs, including normal, extended, disjunctive and nested programs, equilibrium models correspond to answer sets [14,13]. The 'translation' from the syntax of programs to $\mathbf{N}_5$ propositional formulas is the trivial one, eg. a ground rule of an (extended) disjunctive program of the form

$$K_1 \vee \ldots \vee K_k \leftarrow L_1, \ldots, L_m, notL_{m+1}, \ldots, notL_n$$

where the L_i and K_j are literals corresponds to the $\mathbf{N}_5$ sentence

$$L_1 \wedge \ldots \wedge L_m \wedge \neg L_{m+1} \wedge \ldots \wedge \neg L_n \rightarrow K_1 \vee \ldots \vee K_k$$

A set of $\mathbf{N}_5$ sentences is called a *theory*. Two theories are *equivalent* if they have the same equilibrium models.

3 Uniform Equivalence

We recall the definition of uniform equivalence and give a new proof of Theorem 3 of [3] for propositional theories in equilibrium logic. Additional motivation for the study of uniform equivalence and references to previous work is given in [3].

Definition 3. *Two theories Π_1 and Π_2 are said to be* uniform equivalent*, or u-equivalent for short, iff for any (empty or non-empty) set X of literals, $\Pi_1 \cup X$ and $\Pi_2 \cup X$ are equivalent, ie have the same equilibrium models.*

Note that if the theories are logic programs, this means they have the same answer sets.

We begin with some simple terminology. A model $\langle H, T\rangle$ is said to be *incomplete* if it is not total, ie. if $H \subset T$. A model $\langle H, T\rangle$ of a theory Π is said to be *maximal incomplete* (or just *maximal*) if it is incomplete and is maximal among incomplete models of Π under the ordering $\trianglelefteq$. In other words a model $\langle H, T\rangle$ of Π is maximal if for any incomplete

model $\langle H', T\rangle$ of Π, if $H \subset H'$ then $H' = T$. It is clear that if a theory Π is finite and has an incomplete model $\langle H, T\rangle$, then it has a maximal incomplete model $\langle H', T\rangle$ such that $H \subseteq H'$. However maximal models need not exist in the case that Π is an infinite theory. The following is straightforward.

Lemma 1. *If two theories Π_1 and Π_2 are u-equivalent, then they have the same total models.*

Note that theories with the same total models are equivalent in classical logic with strong negation (see Gurevich [7]).

Lemma 2. *If two finite theories Π_1 and Π_2 have the same total models and the same maximal incomplete models, then they are equivalent.*

Proof. Equilibrium models are total models with no incomplete 'submodels'.

Lemma 3. *If two finite theories Π_1 and Π_2 have the same maximal and total models then they are uniform equivalent.*

Proof. From Lemma 2 we have seen that theories Π_1 and Π_2 with the same total and maximal models are equivalent. It remains to show that they are also uniform equivalent. Thus, assume that Π_1 and Π_2 have the same total and maximal models and are therefore equivalent. Suppose for the contradiction that they are not u-equivalent. Then for some set X of literals, $\Pi_1 \cup X$ and $\Pi_2 \cup X$ are not equivalent, say the former has an equilibrium model $\langle T, T\rangle$ that is not an equilibrium model of $\Pi_2 \cup X$. Since Π_1 and Π_2 have the same total models, clearly $\langle T, T\rangle \models \Pi_2 \cup X$. By assumption there is a model $\langle H, T\rangle$ of $\Pi_2 \cup X$ with $H \subset T$. Clearly $X \subseteq H$. Keeping T fixed, extend H to a maximal incomplete model $\langle H', T\rangle$ of Π_2 in T. It is evident that $\langle H', T\rangle$ is not a maximal model (or even model) of Π_1. If it were, since $X \subseteq H'$, it would be an incomplete model of $\Pi_1 \cup X$, contradicting the assumption that $\langle T, T\rangle$ is in equilibrium. Consequently, if two theories are not u-equivalent, they differ on some maximal model, contradicting the initial assumption. □

Lemma 4. *If two finite theories Π_1 and Π_2 are uniform equivalent, then they have the same maximal and total models.*

Proof. By Lemma 1, u-equivalent theories have the same total models. We will show that if they differ on maximal models, then they are not u-equivalent. Thus, suppose that Π_1 and Π_2 differ on some maximal incomplete model. Suppose for instance that Π_1 has a maximal incomplete model $\langle H, T\rangle$ that is not a maximal incomplete model of Π_2. We distinguish two cases as follows. Case (i): there is a model $\langle H', T\rangle$ of Π_2 with $H \subseteq H'$. Case (ii): there is no such model of Π_2. In each case we define non-equivalent extensions of Π_1 and Π_2.

Case (i). Since by assumption $\langle H, T\rangle$ is not a maximal model of Π_2, we can choose H' such that $H \subset H'$. Now select any element $A \in H' - H$. Set $X = H \cup \{A\}$. Then clearly $\langle T, T\rangle \models \Pi_1 \cup X$ and for any model $\langle J, T\rangle$ of $\Pi_1 \cup X$, clearly $H \subset J$. Hence it follows from the maximality of $\langle H, T\rangle$ among models of Π_1 that $\langle T, T\rangle$ must be an equilibrium model of $\Pi_1 \cup X$. But by inspection $\langle H', T\rangle$ is a model of $\Pi_2 \cup X$

and so $\langle T, T\rangle$ is not an equilibrium model of $\Pi_2 \cup X$, showing that Π_1 and Π_2 are not u-equivalent.

Case (ii). There is no model $\langle H', T\rangle$ of Π_2 with $H \subseteq H'$. In this case set $X = H$ and consider $\Pi_1 \cup X$ and $\Pi_2 \cup X$. Since Π_1 and Π_2 have the same total models it is clear that $\langle T, T\rangle \models \Pi_2 \cup X$. Moreover it is an equilibrium model of $\Pi_2 \cup X$ since, by assumption, there is no incomplete model $\langle H', T\rangle$ of Π_2 with $H \subset H'$. But clearly $\langle T, T\rangle$ is not an equilibrium model of $\Pi_1 \cup X$, since $\langle H, T\rangle \models \Pi_1$ and hence $\langle H, T\rangle \models \Pi_1 \cup H$. □

So we have shown Theorem 3 of [3] for arbitrary, finite theories:

Theorem 1. *Two finite theories are uniform equivalent if and only if they have the same total and maximal incomplete models.*

3.1 An Extension

u-equivalence is typically of interest when one has a fixed set of program rules or a deductive database (the intensional part) and a variable set of facts or atomic propositions (or their strong negations) changing over time (the extensional part). Now suppose we allow the extensional part to contain other kinds of formulas, including say disjunctions and integrity constraints. What kinds of equivalences are obtained in such cases? We know from [13] that adding even the simplest kinds of proper rules with implication, of the form $p_i \rightarrow p_j$, brings us to the full case of strong equivalence. The next 'strongest' case would be to allow arbitrary formulas in $(\wedge, \vee, \neg, \sim)$ but without implication.

Definition 4. *Two theories Π_1 and Π_2 are* uniform equivalent in the extended sense*, or u^+-equivalent, iff for any (empty or non-empty) set X of (implication-free) formulas in $(\wedge, \vee, \neg, \sim)$, $\Pi_1 \cup X$ and $\Pi_2 \cup X$ are equivalent, ie have the same equilibrium models.*

It turns out that the two notions are equivalent. To see this we need a simple lemma whose proof is left to the reader.

Lemma 5. *Let φ be any formula in $(\wedge, \vee, \neg, \sim)$. If $\langle H, T\rangle \models \varphi$ and $H \subseteq H' \subseteq T$ then $\langle H', T\rangle \models \varphi$.*

Theorem 2. *Two finite theories are u^+-equivalent if and only if they are u-equivalent.*

Proof. By definition, u-equivalence is a special case of u^+-equivalence. In particular if $\Pi_1 \cup X$ and $\Pi_2 \cup X$ are equivalent for all X comprising implication-free formulas, then they are clearly equivalent for all set of literals X. So u^+-equivalence implies u-equivalence. For the other direction, suppose Π_1 and Π_2 are u-equivalent, then by Lemma 4, they have the same maximal and total models. We show that this implies their u^+-equivalence. We proceed exactly as in the proof of Lemma 3. But for the contradiction we now suppose that for some set X of implication-free formulas, $\Pi_1 \cup X$ and $\Pi_2 \cup X$ are not equivalent, say that $\langle T, T\rangle$ is an equilibrium model of $\Pi_1 \cup X$ but not of $\Pi_2 \cup X$. Again since they have the same total models we know that $\langle T, T\rangle \models \Pi_2 \cup X$, but since it is not an equilibrium model, there is a model $\langle H, T\rangle$ of $\Pi_2 \cup X$ with $H \subset T$. Let $\langle H', T\rangle$ be any maximal model of Π_2 such that $H \subset H'$. We need only check that $\langle H', T\rangle \models X$. But this follows immediately from Lemma 5 above. So, as before, $\langle H', T\rangle \not\models \Pi_1$, since $\langle T, T\rangle$ is an equilibrium model of $\Pi_1 \cup X$. □

This result shows that on finite programs uniform and strong equivalence are the only concepts of their kind in ASP. In particular, varying the logical form of formulas permitted in the extension X of a program does not produce any new notion of equivalence: strong equivalence is the appropriate concept when proper rules containing implication are permitted, while all other cases are covered by uniform equivalence.

4 Many-Valued Semantics for $\mathbf{N_5}$

The Kripke semantics for $\mathbf{N}_5$ logic can be easily characterised using a many-valued approach, specifically with a five-valued logic. In this section we define this interpretation and then describe a five-valued tableau system to check inference.

The set of truth values in the many-valued characterisation is $\mathbf{5} = \{-2, -1, 0, 1, 2\}$ and 2 is the designated value. The connectives are interpreted as follows: $\wedge$ is the minimum function, $\vee$ is the maximum function, $\sim x = -x$,

$$x \to y = \begin{cases} 2 & \text{if either } x \leq 0 \text{ or } x \leq y \\ y & \text{otherwise} \end{cases} \qquad \text{and} \qquad \neg x = \begin{cases} 2 & \text{if } x \leq 0 \\ -x & \text{otherwise} \end{cases}$$

Any $\mathbf{N}_5$ model σ as a truth-value assignment can trivially be converted into a Kripke model $\langle H, T\rangle$, and *vice versa*. For example, if σ is an assignment and p is a propositional variable, then the corresponding Kripke model, denoted by $\mathcal{M}_\sigma$, is determined by the equivalences:

$$\begin{array}{llll} \sigma(p) = 2 & \text{iff} & p \in H & \qquad \sigma(p) = -1 \quad \text{iff} \quad \sim p \in T, \sim p \notin H \\ \sigma(p) = 1 & \text{iff} & p \in T, p \notin H & \qquad \sigma(p) = -2 \quad \text{iff} \quad \sim p \in H \\ \sigma(p) = 0 & \text{iff} & p \notin T, \sim p \notin T & \end{array}$$

The many-valued semantics and the Kripke semantics for $\mathbf{N}_5$ are equivalent. In other words, if Π is a set of formulas in $\mathbf{N}_5$ and ψ is a formula, then $\Pi \models \psi$ iff for every assignment σ in $\mathbf{N}_5$, if $\sigma(\varphi) = 2$ for every $\varphi \in \Pi$, then $\sigma(\psi) = 2$. Note too that assignments or truth-value interpretations can also be considered partially ordered by the $\trianglelefteq$ relation. We then say for example that an assignment σ is greater than or equal to an assignment τ, if $\mathcal{M}_\tau \trianglelefteq \mathcal{M}_\sigma$.

4.1 Tableau Systems for $\mathbf{N_5}$

In [15] we introduced tableaux systems to study several properties in $\mathbf{N}_5$. Specifically, one system to check validity, another one to generate total models and another system based on auxiliary tableaux to check the equilibrium property for a specific model.[3] The systems are described using *signed-formulas*, following the approach of [8]. The formulas in the tableau systems are labelled with sets of truth values: $S{:}\varphi$, $S \subset \mathbf{5}$; an assignment σ in $\mathbf{N}_5$ is a model of $S{:}\varphi$ if $\sigma(\varphi) \in S$. The initial tableau determines the goal of the system; to study the satisfiability of a set of formulas $\{\varphi_1, \ldots, \varphi_n\}$ and generate its models we use the following:

[3] Note that the tableaux system for $\mathbf{N}_5$ is already adequate for checking strong equivalence.

Initial tableau for satisfiability $\begin{cases} \{2\}{:}\varphi_1 \\ \dots \\ \{2\}{:}\varphi_n \end{cases}$

So we look for assignments, σ such that $\sigma(\varphi_i) = 2$ for all i. The **expansion rules** are common for every system and they must comply with the following property: *a model for any branch must be a model for the initial tableau and every model of the initial tableau is a model of some branch*. We show the rules for the connective $\rightarrow$ in Figure 1, the other connectives, $\wedge$, $\vee$, $\sim$ and $\neg$ are *regular* connectives, and the standard expansion rules can be applied [8].

1. If T is a tableau and T' is the tree obtained from T applying one of the expansion rules, then T' is a tableau. As usual in tableau systems for propositional logics, if a formula can be used to expand the tableau, then the tableau is expanded in every branch below the formula using the corresponding rule and the formula used to expand is marked and is no longer used.
2. A branch B in a tableau T is called *closed* if it contains a variable p with two signs, $S{:}p$, $S'{:}p$, such that $S \cap S' = \varnothing$, that is, the branch is unsatisfiable.
3. A branch B in a tableau T is called *finished* if the non-marked formulas are labelled propositional variables, *signed literals*. The branch is called *open* if it is non-closed and finished; in this case every model of the set of signed literals is a model of the initial set of formulas.
4. A tableau T is called *closed* if every branch is closed; in this case the initial set of formulas is unsatisfiable. The tableau is *open* if it has an open branch, (ie if it is non-closed). And it is *terminated* if every branch is either closed or open.

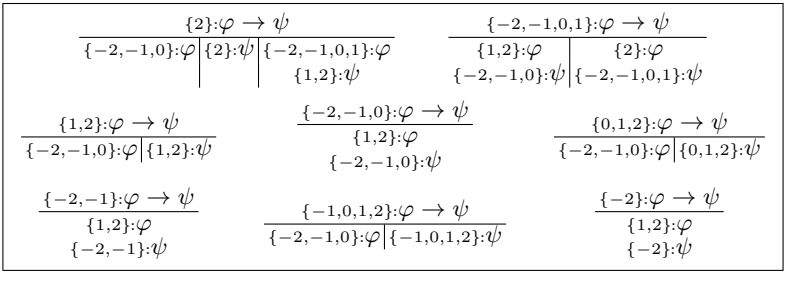


Fig. 1. Tableau expansion rules in $\mathbf{N}_5$ for $\rightarrow$

In the system, we look for non-closed terminated tableaux, that allows us to generate all the models of the initial set of formulas.

Theorem 3. *Let T be a non-closed terminated tableau for $\Pi = \{\varphi_1, \dots, \varphi_n\}$. Then, σ is a model of Π if and only if some branch of T is satisfiable by σ.*

4.2 Uniform Equivalence

In the tableaux system for checking u^+-equivalence the following characterisation will be used. It is important to recall that it is valid also for infinite theories.

Theorem 4. *Two theories Π_1 and Π_2 with the same total models are u^+-equivalent if and only if the following conditions hold:*

(a) If $\langle H, T\rangle$ is an incomplete model of Π_1 then there exists H' such that $H \subseteq H' \subset T$ and $\langle H', T\rangle$ is a model of Π_2.
(b) If $\langle H, T\rangle$ is an incomplete model of Π_2 then there exists H' such that $H \subseteq H' \subset T$ and $\langle H', T\rangle$ is a model of Π_1.

Proof: ($\Leftarrow$) Assume that conditions (a) and (b) hold and Π_1 and Π_2 are not u^+-equivalent; let us assume that there exists a set of implication-free formulas, X, and an incomplete interpretation $\langle H, T\rangle$ such that $\langle T, T\rangle$ is an equilibrium model of $\Pi_1 \cup X$ and $\langle H, T\rangle \models \Pi_2 \cup X$. Obviously, $\langle H, T\rangle \models \Pi_2$ and $\langle H, T\rangle \models X$ and, by condition (b), there exists H' such that $H \subseteq H' \subset T$ and $\langle H', T\rangle \models \Pi_1$. Moreover, by lemma 5, $\langle H', T\rangle \models X$ and thus $\langle H', T\rangle \models \Pi_1 \cup X$, contradicting the equilibrium property of $\langle T, T\rangle$.

($\Rightarrow$) Assume that Π_1 and Π_2 are u^+-equivalent. We show (a); the proof for (b) is similar. Let H and T be such that $H \subset T$ and $\langle H, T\rangle \models \Pi_1$ and assume that for every H' such that $H \subseteq H' \subset T$ we have $\langle H', T\rangle \not\models \Pi_2$. Obviously, $\langle T, T\rangle \models H$, $\langle T, T\rangle \models \Pi_1$, thus $\langle T, T\rangle \models \Pi_2$, and $\langle T, T\rangle \models \Pi_2 \cup H$. Moreover, every model of H must be between $\langle H, T\rangle$ and $\langle T, T\rangle$ and none of them is a model of Π_2; so, there is no model for $\Pi_2 \cup H$ less than $\langle T, T\rangle$ and therefore this is in equilibrium. Since Π_1 and Π_2 are u^+-equivalent, $\langle T, T\rangle$ is also an equilibrium model of $\Pi_1 \cup H$, contradicting that $\langle H, T\rangle \models \Pi_1 \cup H$. □

This result is given for the case of disjunctive logic programs as Theorem 1 of [3]; note however that they state the conditions in (a) and (b) as 'iff' rather than 'if-then' conditions.

Auxiliary tableaux for u^+-equivalence checking. An algorithm for u^+-equivalence checking is sketched as follows

1. The models of Π_1 and Π_2 are generated using the tableaux method in section 4.1.
2. If either some total model of Π_1 is not a model of Π_2 or some total model of Π_2 is not a model of Π_1, then Π_1 and Π_2 are not u^+-equivalent.
3. If Π_1 and Π_2 have the same total models, then we check if every incomplete model of Π_1 has a greater interpretation that is a model for Π_2, and also if every incomplete model of Π_2 has a greater interpretation that is a model for Π_1. For this we use auxiliary tableaux that are constructed for Π_1 and every incomplete model of Π_2, and for Π_2 and every incomplete model of Π_1.

The goal of the auxiliary tableau for a theory Π and an interpretation σ is to check if there is an incomplete model for Π greater than σ. For that, we add to the initial tableau a set of signed literals built from σ:

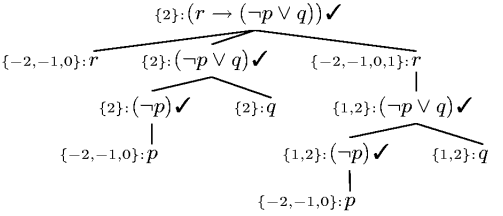
$$\{(S \cap \omega_\sigma(p)){:}p; p \text{ atom in } \Pi\}$$

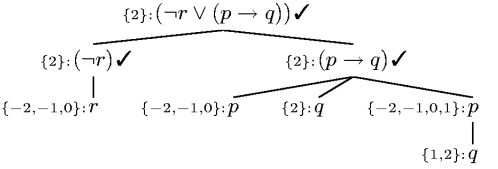
where ω_σ is defined by: $\omega_\sigma(p) = \{\sigma(p)\}$ if $\sigma(p) \in \{-2, 0, 2\}$, $\omega_\sigma(p) = \{-2, -1\}$ if $\sigma(p) = -1$ and $\omega_\sigma(p) = \{1, 2\}$ if $\sigma(p) = 1$. With these signed formulas we restrict the admisible models for a branch to those that are greater than σ.

Theorem 5. *Given a theory Π and an assignment σ, there exists an incomplete model of Π greater than σ if and only if there is a tableau for (Π, σ) and an open branch with an incomplete model.*

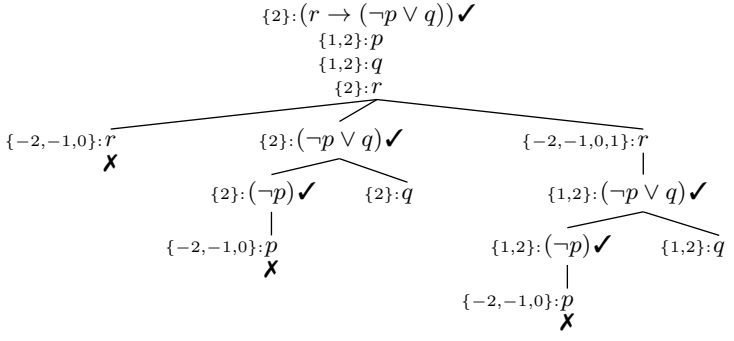
Example: We are going to check that $\varphi_1 = r \to (\neg p \vee q)$ and $\varphi_2 = \neg r \vee (p \to q)$ are u^+-equivalent. The tableaux on the right allow us to generate the models of φ_1 and φ_2.

Comparing the sets of models, it is easy to conclude that the total models are the same for both formulas and we obtain two special models, σ and τ, defined as follows: $\sigma(p) = 2$, $\sigma(q) = 1$, $\sigma(r) = 1$, $\tau(p) = 1$, $\tau(q) = 1$, $\tau(r) = 2$; or, equivalently, $\sigma = \langle\{p\}, \{p, q, r\}\rangle$, $\tau = \langle\{r\}, \{p, q, r\}\rangle$.

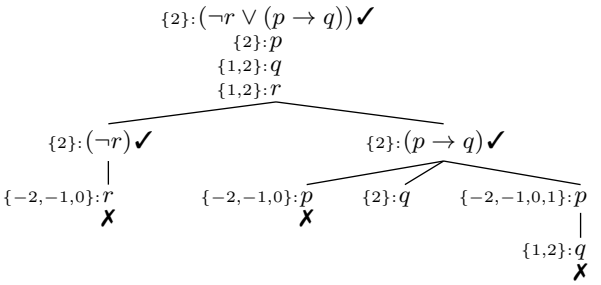




These interpretations verify: $\sigma \models \varphi_1$, $\tau \models \varphi_2$, $\sigma \not\models \varphi_2$, $\tau \not\models \varphi_1$. Thus, φ_1 and φ_2 are not strongly equivalent, however they are uniformly equivalent. To see this we construct the auxiliary tableaux for (φ_1, τ) and for (φ_2, σ). In the previous tableaux for φ_1 we add the literals: $\{1,2\}{:}p$, $\{1,2\}{:}q$ and $\{2\}{:}r$ and the resulting auxiliary tableau for (φ_1, τ) remains open with incomplete models:



For (φ_2, σ) the auxiliary tableau also remains open with incomplete models:



Similar examples will be discussed below. In [3] the complexity of uniform equivalence checking is studied. For the restriction to disjunctive programs the conclusion is that the problem is Π_2^P-complete. Our procedure based on tableau systems allow us to conclude that the problem for general theories is also Π_2^P-hard, because the generation of models is coNP-hard and the maximality checking (the oracle) is NP-hard. Therefore, the problem of uniform equivalence checking for general theories is also Π_2^P-complete.

5 Examples

1. We can consider examples of disjunctive programs allowing negation in the head such as
$$\Pi_1 = \{r \to (\neg p \vee q)\}$$
It is readily seen that Π_1 is not strongly equivalent to
$$\Pi_2 = \{(r \wedge p) \to q\}$$
The interpretation $\langle\{r\}, \{p, q, r\}\rangle$ is a model of Π_2 but not of Π_1 (and is the only distinguishing model), while $\langle\{q, r\}, \{p, q, r\}\rangle$ is a model of Π_1. Therefore Π_1 and Π_2 do not differ on maximal incomplete models and are therefore uniform equivalent.
Notice however that it is not the case that $\{r \to (p \vee q)\} \equiv_u \{(r \wedge \neg p) \to q\}$, since for example $\langle\{r\}, \{p, r\}\rangle \models \{(r \wedge \neg p) \to q\}$, but $\langle\{r\}, \{p, r\}\rangle \not\models \{r \to (p \vee q)\}$.
2. A distinguishing feature of equilibrium logic is that it allows us to express not only programs with nested expressions (as above) in the style of [12] but also theories or programs with nested rules, eg programs with conditional rules containing implication in the body, of the form $(p \to q) \to r$. It is interesting to consider when such rules can be replaced by "ordinary" nested rules. Consider for instance:
$$\Pi_1 = \{((p \to q) \wedge s) \to r\}$$
versus
$$\Pi_2 = \{((\neg p \vee q) \wedge s) \to r\}$$
It is easy to see that these formulas are equivalent but not strongly equivalent and therefore not interchangeable in all contexts. In particular $\langle\{s\}, \{p, q, r, s\}\rangle \models \Pi_2$,

but $\langle\{s\},\{p,q,r,s\}\rangle \not\models \Pi_1$. This is the only interpretation that distinguishes the two formulas but it is not a maximal model of Π_2, since $\langle\{p,s\},\{p,q,r,s\}\rangle$ is also a Π_2 model. Consequently the formulas are uniform equivalent.

3. We may also consider the case where implication is permitted in the heads of rules. But one can easily see that in the simple case of $r \rightarrow (p \rightarrow q)$ there is strong equivalence with respect to the normal rule $(r \wedge p) \rightarrow q$. In fact one can see that in this respect equilibrium logic differs from the semantics for nested rules proposed by [6].

Since we include programs with nested expressions, the uniform equivalence for programs with weight constrains and choice rules can be similar treated thanks to the connection established in [4].

6 Conclusions and Future Work

The uniform equivalence of logic programs is an important property that may, in specific contexts, allow one program to be substituted by another, perhaps syntactically simpler, program. In the general case, however, it is a harder property to check than strong equivalence. Here we have outlined a proof system for verifying uniform equivalence that applies to general propositional theories and consequently to any selected subclass of logic programs. The system is based on a semantical characterisation of uniform equivalence, similar to that of Eiter and Fink [3], but formulated for general propositional theories in terms of ordinary models in the logic $\mathbf{N}_5$ of here-and-there with strong negation. As a by-product we were able to show that varying the logical form of formulas in the program extensions does not change the properties of uniform equivalence, and hence there are no other types of equivalence of this kind situated 'between' uniform and strong equivalence.

We already remarked that uniform equivalence may be relevant in application areas of answer set programming, such as diagnosis and planning, where abductive methods are used. It is clear that if all and only atoms are permitted as abducibles, then uniform equivalent programs have the same (abductive) explanatory power, since every abductive explanation in one program is an equivalent explanation in the other.[4] Our results show that this is still the case when abducibles are allowed to be any implication-free formulas. However there appears to be a useful concept of abductive equivalence weaker than uniform equivalence. Two programs Π_1 and Π_2 would be equivalent in this weaker sense if for any X there exists a Y such that $\Pi_1 \cup X$ and $\Pi_2 \cup Y$ are equivalent and vice versa; X, Y are suitably restricted (but possibly different) sets of formulas (eg literals). It remains to be seen whether this concept is genuinely weaker and whether it can be characterised in simple, semantic terms.

Efficiency issues also arise from our basic algorithms. The exhaustive generation of models can be computationally hard; in the example displayed in Section 4, each formula has 111 distinct models, though ultimately we only need to manipulate two models. So, further refinements of the algorithm remain be investigated in the future [2,1].

[4] We are assuming here the absence of additional syntactic restrictions such as minimality conditions.

References

1. G. Aguilera Venegas, I. Perez de Guzman, M. Ojeda Aciego, and A. Valverde. Reducing signed propositional formulas. *Soft Computing—A Fusion of Foundations, Methodologies and Applications*, 2(4):157–166, 1999.
2. I.P. de Guzmán, M. Ojeda-Aciego, and A. Valverde. *Multiple-Valued Tableaux with Δ-reductions*. In *Proceedings of IC-AI'99*, pages 177–183, 1999.
3. T. Eiter and M. Fink. Uniform equivalence of logic programs under the stable model semantics. In *Int. Conf. in Logic Programming, ICLP'03*, Mumbay, India. Springer, 2003.
4. P. Ferraras and V. Lifschitz Weight Constraints as Nested Expressions. *Theory and Practice of Logic Programming*, (to appear).
5. K. Gödel. Zum intuitionistischen aussagenkalkül. *Anzeiger der Akademie der Wissenschaften Wien, mathematisch, naturwissenschaftliche Klasse*, 69:65–66, 1932.
6. S. Greco, N. Leone and F. Scarcello, Disjunctive Datalog with Nested Rules, in J. Dix *et al* (eds), *Logic Programming and Knowledge Representation. Proc. LPKR97*, Spinger, LNCS 1471, 1998.
7. Y. Gurevich. Intuitionistic logic with strong negation. *Studia Logica*, 36(1–2):49–59, 1977.
8. R. Hähnle. *Automated Deduction in Multiple-Valued Logics*. Oxford University Press, 1993.
9. A. Heyting. Die formalen regeln der intuitionistischen logik. *Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse*, pages 42–56, 1930.
10. D. De Jongh and L. Hendriks. Characterization of strongly equivalent logic programs in intermediate logics. *Theory and Practice of Logic Programming*, 3(3):259–270, 2003.
11. M. Kracht. On extensions of intermediate logics by strong negation. *Journal of Philosophical Logic*, 27(1):49–73, 1998.
12. V. Lifschitz, L. Tang and H Turner, Nested Expressions in Logic Programs, *Annals of Mathematics and Artificial Intelligence*, 25, 369–389, 1999.
13. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
14. D. Pearce. A new logical characterisation of stable models and answer sets. In *Non-Monotonic Extensions of Logic Programming, NMELP 96*, LNCS 1216, pages 57–70. Springer, 1997.
15. D. Pearce, I.P. de Guzmán, and A. Valverde. A tableau calculus for equilibrium entailment. In *Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2000*, LNAI 1847, pages 352–367. Springer, 2000.
16. H. Turner. Strong equivalence for logic programs and default theories (made easy). In *Proc. of the Logic Programming and Nonmonotonic Reasoning, LPNMR'01*, LNAI 2173, pages 81–92. Springer, 2001.
17. D. van Dalen. Intuitionistic logic. In *Handbook of Philosophical Logic, Volume III: Alternatives in Classical Logic*, Dordrecht, 1986. D. Reidel Publishing Co.
18. N. N. Vorob'ev. A constructive propositional calculus with strong negation (in Russian). *Doklady Akademii Nauk SSR*, 85:465–468, 1952.
19. N. N. Vorob'ev. The problem of deducibility in constructive propositional calculus with strong negation (in Russian). *Doklady Akademii Nauk SSR*, 85:689–692, 1952.

Partial Stable Models for Logic Programs with Aggregates

Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe

Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{pelov,marcd,maurice}@cs.kuleuven.ac.be

Abstract. We introduce a family of partial stable model semantics for logic programs with arbitrary aggregate relations. The semantics are parametrized by the interpretation of aggregate relations in three-valued logic. Any semantics in this family satisfies two important properties: (i) it extends the partial stable semantics for normal logic programs and (ii) total stable models are always minimal. We also give a specific instance of the semantics and show that it has several attractive features.

1 Introduction

Aggregates are an important concept for natural modeling of many problems. Existing work already covers a large class of aggregate programs, like monotone [13,15] and stratified [1,13] ones. There are, however, programs which involve recursion over non-monotone aggregation and do not fall in any of these classes. An example is the Party Invitation problem (Example 1). The development of a uniform semantics of non-monotone aggregation is also important in the context of the Answer Set Programming paradigm where logic programming under the stable model semantics is used as a knowledge representation language.

In this paper we define partial stable models for logic programs with arbitrary aggregate relations. We point out that the well-founded model and the total stable models are special types of partial stable models. Thus, our definition also covers these two semantics.

The foundation of this work is the algebraic theory of approximating operators developed by Denecker, Marek, and Truszczyński [3,4]. The theory studies approximations of the fixpoints of non-monotone lattice operators $O : L \to L$. With any such operator O, it associates a family of approximating operators $A : L^2 \to L^2$ on the product lattice L^2. Every operator A determines a set of partial stable fixpoints. In logic programming we are interested in approximating the fixpoints of the T_P operator. One possible approximating operator of T_P is the standard four-valued immediate consequence operator Φ_P [7]. The partial stable fixpoints of Φ_P, as defined by Approximation Theory [3], correspond with partial stable models as defined by Przymusinski [14] and Fitting [7]. Recently, Denecker et al. [4] have investigated the semantics determined by another approximating operator of T_P — the *ultimate* approximating operator U_P. This

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 207–219, 2004.

is the most precise approximating operator of T_P. Consequently, it has a more precise well-founded fixpoint and a larger number of total stable fixpoints than any other approximating operator of T_P.

To define partial stable model semantics for aggregate programs we have to define an approximating operator of the two-valued immediate consequence operator T_P^{aggr} for aggregate programs. In [5], we studied the semantics defined by the ultimate approximating operator U_P^{aggr} of T_P^{aggr}. For logic programs without aggregates the U_P^{aggr} operator is the same as the U_P operator. We showed that the ultimate well-founded and ultimate stable semantics correctly model the intended meaning of many typical aggregate programs.

One concern with the ultimate semantics is that for logic programs without aggregates it does not coincide with the standard well-founded and stable semantics [4]. Another concern is the higher computational complexity — it goes one level higher in the polynomial hierarchy than the complexity of the standard semantics [4]. In this paper we define an alternative approximating operator of T_P^{aggr}. Our goal is to extend the standard partial stable semantics of normal logic programs [14]. We achieve this by extending the standard approximating operator Φ_P to an approximating operator Φ_P^{aggr} for logic programs with aggregates. Because the partial stable semantics for logic programs, and in particular the well-founded and total stable models, are widely accepted semantics we argue that extending these semantics is an important feature of our work.

One of the contributions of the paper is giving precise relationship between our semantics and most of the previous proposals for stable model semantics for aggregate programs. This includes the stable semantics of weight constraint rules [16] used by the smodels system, the stable semantics of Kemp and Stuckey [11] which is also used by A-Prolog [8] and the dlv system [2], and our previous work on the ultimate semantics of aggregate programs [5].

The structure of the paper is as follows. We start by defining the syntax and semantics of aggregate programs (Section 2). In Section 3 we give the definition of partial stable models. In Section 4 we study the relationship between our semantics and other proposals. Finally, we give some comments on current and future work (Section 5).

2 Logic Programs with Aggregates

In our work on the ultimate semantics of aggregate programs [5], an aggregate relation is an arbitrary 2-nd order predicate. In this work we use a more restricted and more conventional notion of an aggregate as a function or relation on finite multisets. A multiset (also called a *bag*) is similar to a set except that an object can occur multiple times. The most general definition of a multiset M on a domain D is a mapping $M : D \to Card$ where $Card$ is the class of cardinal numbers. For every element $x \in D$, $M(x)$ gives the multiplicity of x. In this work we consider only finite multisets. First, we define a *multiset with finite multiplicity* as a function $M : D \to \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers. A *finite multiset* is a multiset with finite multiplicity such that $M(x) > 0$ only for

a finite number of elements. We denote the set of all finite multisets on D with $\mathcal{M}(D)$. The subset relation between multisets is defined as follows: $M_1 \subseteq M_2$ if and only if $M_1(x) \leq M_2(x)$ for all $x \in D$. The *additive union* $M = M_1 \uplus M_2$ of M_1 and M_2 is defined as $M(x) = M_1(x) + M_2(x)$ for all $x \in D$. The *multiset difference* $M = M_1 - M_2$ of M_1 and M_2 is defined as $M(x) = M_1(x) - M_2(x)$ if $M_1(x) \geq M_2(x)$ and $M(x) = 0$ otherwise. We denote the *empty multiset* with $\emptyset$. For the rest of the paper by multiset we mean a finite multiset.

An *aggregate function* is any function $\mathrm{F} : \mathcal{M}(D_1) \to D_2$. Examples of aggregate functions are $\textsc{card} : \mathcal{M}(D) \to \mathbb{N}$ for any set D returning the number of elements in the multiset and $\textsc{sum} : \mathcal{M}(\mathbb{R}) \to \mathbb{R}$ returning the sum of the elements in the multiset. Both functions are well-defined because we consider only finite multisets. For simplicity of the presentation we assume that the input multiset D_1 and the domain of the result D_2 of aggregate functions are equal.

Some aggregate functions are not well-defined on all multisets. One example is taking the average of the empty set. One way to model for such behavior is to allow aggregate functions to be partial. However, a more general solution is to consider not only aggregate functions but also aggregate relations. An *aggregate relation* on D is any relation $\mathrm{R} \subseteq \mathcal{M}(D) \times D$. Under this view, an aggregate can also return a set of elements.

One advantage of representing aggregates as relations is that we can obtain new aggregate relations by composition of an existing aggregate with another relation. Let R be an aggregate relation on D and $P \subseteq D \times D$ a binary relation on D. The *composition* of R and P, denoted as R_P, is an aggregate relation defined as follows: $(M, d) \in \mathrm{R}_P$ if and only if there exists $a \in D$ such that $(M, a) \in \mathrm{R}$ and $(a, d) \in P$. Typically, the binary relation P is some partial order relation on the domain D. For example, the $\textsc{sum}_{\geq}$ aggregate relation means that the sum of the elements in the multiset is greater than or equal to the second argument.

Monotonicity and anti-monotonicity of aggregate relations is defined with respect to the input multiset.

Definition 1. *Let* R *be an aggregate relation on* D*. We say that* R *is:*

- monotone *if* $(M_1, d) \in \mathrm{R}$ *and* $M_1 \subseteq M_2$ *implies* $(M_2, d) \in \mathrm{R}$*;*
- anti-monotone *if* $(M_2, d) \in \mathrm{R}$ *and* $M_1 \subseteq M_2$ *implies* $(M_1, d) \in \mathrm{R}$*.*

2.1 Set Expressions, Aggregate Atoms, and Aggregate Programs

Aggregate programs are built over a set of propositional atoms At and a domain D. A *literal* is an atom a (*positive literal*) or the negation of an atom *not* a (*negative literal*). The *complement* $\overline{L}$ of a literal L is defined as $\overline{L} = \mathit{not}\ a$ if $L = a$ and $\overline{L} = a$ if $L = \mathit{not}\ a$.

A *weight literal* is an expression $L = w$ where L is a literal and $w \in D$ is a weight associated with L. A *set expression* is a finite set of weight literals. Syntactically, we denote a set expression as $\{L_1 = w_1, \ldots, L_n = w_n\}$. For a set expression s, $w(s)$ denotes the multiset consisting of the weights of all weight literals in s.

An *aggregate atom* has the form $\mathrm{R}(s, d)$ where R is an aggregate relation, s is a set expression, and $d \in D$. An example of an aggregate atom is $\mathrm{SUM}_{\geq}(\{a = 1, b = 2\}, 2)$. It is true in an interpretation I if the aggregate relation is true for the multiset consisting of the weights of the literals true in I, i.e. if the sum of those weights is greater than or equal to 2.

A *rule* has the form $A \leftarrow B_1 \wedge \ldots \wedge B_n$ where A is an atom called the *head* of the rule and every B_i is a literal or an aggregate atom. An *aggregate program* is a (possibly infinite) set of rules. A program without aggregate atoms is called *normal logic program.*

Definition 2. *A* positive (negative) aggregate atom *is an aggregate atom* $\mathrm{R}(s, d)$ *such that either* R *is a monotone aggregate relation and s contains only positive (negative) weight literals, or* R *is an anti-monotone aggregate relation and s contains only negative (positive) weight literals.*

A definite aggregate program *is a program which contains only positive literals and positive aggregate atoms.*

Satisfiability of positive aggregate atoms is monotone and satisfiability of negative aggregate atoms is anti-monotone. If all aggregate atoms fall in one of the two classes then one can easily define semantics which treats aggregate atoms as positive and negative literals respectively. There are, however, aggregate atoms whose satisfiability is neither monotone nor anti-monotone. This fact requires a more elaborate treatment of aggregate atoms.

To illustrate the syntax of aggregate programs we give one version of the party invitation problem [17, Example 6.3]. This is an interesting example because it involves recursion over non-monotone aggregate relations and cannot be handled by some of the previous work on aggregates, e.g. stratified aggregation by Mumick et al. [13] or monotonic aggregation by Ross and Sagiv [15].

Example 1 (Party Invitation). A set of people S are invited to a party. Every person p has a set of compatibility measures w_{pq} with some other people. This measure can be both positive and negative. In the latter case this expresses the fact that p dislikes q. A person p will accept the invitation if the sum of the compatibilities with all other people who accept the invitation is greater than or equal than some threshold k_p. For every person p we have a rule

$$a_p \leftarrow \mathrm{SUM}_{\geq}(\{a_{q_1} = w_{pq_1}, \ldots, a_{q_m} = w_{pq_m}\}, k_p).$$

where $q_1, \ldots, q_m$ are all persons with whom p has a non-zero compatibility measure, i.e. $w_{pq_i} \neq 0$ for all $i = 1, \ldots, m$.

Consider a situation where a person a will accept the invitation if and only if b does and b will accept if and only if a does not. The precise input is as follows:

$$S = \{a, b\} \qquad w_{ab} = 1 \qquad k_a = 1$$
$$w_{ba} = -1 \qquad k_b = 0$$

and the program is:

$$a \leftarrow \mathrm{SUM}_{\geq}(\{b = 1\}, 1).$$
$$b \leftarrow \mathrm{SUM}_{\geq}(\{a = -1\}, 0).$$

An interpretation for an aggregate program P is defined as the set of atoms which are assigned the value true. The set of all interpretations is denoted with $\mathcal{I}$. It forms a complete lattice under the standard set inclusion order $\subseteq$. Satisfiability of a literal L by an interpretation I is defined in the usual way and denoted with $I \models L$.

For a set expression s and an interpretation I we denote with s^I the subset expression of s which contains only the literals which are true in I that is $s^I = \{L = w \in s \mid I \models L\}$. We now define an evaluation function for set expressions as $[\![s]\!]_I = w(s^I)$ that is the multiset of weights of all literals in s which are true in I. Satisfiability of an aggregate atom $\text{R}(s, d)$ is defined as: $I \models \text{R}(s, d)$ if and only if $([\![s]\!]_I, d) \in \text{R}$. For example, $\{b\} \models \text{SUM}_{\geq}(\{a = 1, b = 2\}, 2)$ while $\{a\} \not\models \text{SUM}_{\geq}(\{a = 1, b = 2\}, 2)$.

With every logic program with aggregates P we can associate an immediate consequence operator T_P^{aggr} in an obvious way.

Definition 3. $T_P^{aggr}(I) = \{A \mid A \leftarrow B \in P \text{ and } I \models B\}$

Proposition 1. *If P is a definite aggregate program then T_P^{aggr} is monotone.*

3 Partial Stable Models

3.1 Approximation Theory

We now present the necessary background on Approximation Theory. The theory was first developed for bilattices (which, in logic programming, correspond to four-valued logic and four-valued interpretations) [3]. In a later work [4] the authors have shown that consistent (three-valued) stable fixpoints depend entirely on the restriction of approximating operators to consistent approximations (three-valued interpretations). The present work is based on this second version of approximation theory which we present below. The reason for this is that it was not clear to us how to define satisfiability of aggregate atoms in four-valued logic while this is possible to do in three-valued logic.

The basic concept in approximation theory is that of an approximation of the elements of a lattice $\langle L, \leq \rangle$ by pairs (x, y). If $x \leq y$ then we call the pair (x, y) *consistent* and if $x = y$ then we call the pair *exact*. We denote the set of all consistent pairs on L with L^c. We call the elements in the set L^c *partial elements*. For example if L is a set of interpretations then the elements of L^c are called partial interpretations. The partial order relation $\leq$ on L can be extended in two ways to partial orders on L^c:

truth order: $(x, y) \leq_t (x_1, y_1)$ if and only if $x \leq x_1$ and $y \leq y_1$
precision order: $(x, y) \leq_p (x_1, y_1)$ if and only if $x \leq x_1$ and $y_1 \leq y$

A consistent pair (x, y) can be seen as an approximation of all elements in the interval $[x, y] = \{z \in L \mid x \leq z \leq y\}$ which is always non-empty. In this sense, the precision order $\leq_p$ corresponds to the precision of the approximation, that

is $(x, y) \leq_p (x_1, y_1)$ if and only if $[x, y] \supseteq [x_1, y_1]$. Exact pairs (x, x) approximate a single element x and represent the embedding of L in L^c.

Consider the lattice $\mathcal{TWO} = \{\mathbf{f}, \mathbf{t}\}$ of classical truth values ordered as $\mathbf{f} < \mathbf{t}$. We denote the set of consistent elements $\mathcal{TWO}^c$ of $\mathcal{TWO}$ with $\mathcal{THREE}$. The exact pairs $(\mathbf{f}, \mathbf{f})$ and $(\mathbf{t}, \mathbf{t})$ are the embedding of the classical truth values $\mathbf{f}$ and $\mathbf{t}$ respectively and are maximal in the precision order. Only the pair $(\mathbf{f}, \mathbf{t})$ approximates more than one truth value, namely the set $\{\mathbf{f}, \mathbf{t}\}$, and corresponds to the value of *undefined*, denoted with $\mathbf{u}$. The truth order $\leq_t$ is used to define conjunction and disjunction in $\mathcal{THREE}$ which are interpreted as greatest lower bound $\wedge_t$ and least upper bound $\vee_t$ respectively. Negation in $\mathcal{THREE}$ is defined as $\neg(x, y) = (\neg y, \neg x)$. In particular, $\neg\mathbf{f} = \mathbf{t}$, $\neg\mathbf{t} = \mathbf{f}$, and $\neg\mathbf{u} = \mathbf{u}$. This interpretation of the connectives $\neg$, $\wedge$, and $\vee$ is the same as the one given by Kleene's strong three-valued logic.

We briefly recall the definition of partial stable operator, the definition of different classes of stable fixpoints, and some basic properties, following [4].

Definition 4 (Partial Approximating Operator). *Let $O : L \to L$ be an operator on a complete lattice L. We say that $A : L^c \to L^c$ is a* partial approximating operator *of O if the following conditions are satisfied:*

- *A* extends *O, i.e. $A(x, x) = (O(x), O(x))$ for every $x \in L$;*
- *A is $\leq_p$-monotone.*

Because A is a $\leq_p$-monotone operator on a complete semilattice it has a least fixpoint called the *Kripke-Kleene* fixpoint of A.

We denote the projection of an approximating operator $A : L^c \to L^c$ on the first and second components with A^1 and A^2, i.e. if $A(x, y) = (u, v)$ then $A^1(x, y) = u$ and $A^2(x, y) = v$. From the $\leq_p$-monotonicity of A follows that A^1 is monotone in the first argument and A^2 is monotone in the second argument.

Recall that for a fixed element $b \in L$, the operator $A^1(\cdot, b)$ is defined only on the domain $[\bot, b]$. However, it is possible that for some element $x \in [\bot, b]$, $A^1(x, b) \notin [\bot, b]$. Similarly, for a fixed element $a \in L$, the operator $A^2(a, \cdot)$ is defined on the domain $[a, \top]$ but it is possible that for some element $y \in [a, \top]$, $A^2(a, y) \notin [a, \top]$. Denecker et al. [4] showed that if (x, y) is a post-fixpoint of A, i.e. $(a, b) \leq_p A(a, b)$ then the operators $A^1(\cdot, b)$ and $A^2(a, \cdot)$ are well-defined on the domains $[\bot, b]$ and $[a, \top]$ respectively. Such pairs (a, b) are called *A-reliable*. Let L^r denote the set of A-reliable pairs. The *partial stable operator* $S : L^r \to L^c$ is defined as follows:

$$S(a, b) = (\, lfp(\lambda x.\ A^1|_{[\bot,b]}(x, b)),\\ lfp(\lambda y.\ A^2|_{[a,\top]}(a, y))\,).$$

Fixpoints of S are called *partial stable fixpoints* of A. The set $ST(A)$ of *exact stable fixpoints* of A is defined as $ST(A) = \{x \in L \mid (x, x) \text{ is a fixpoint of } S\}$. An important property of exact stable fixpoints is that they are minimal fixpoints of O. For such fixpoints it is possible to give a simpler characterization: x is an exact stable fixpoint if and only if $O(x) = x$ and $lfp(A^1(\cdot, x)) = x$. Finally,

the stable operator S is $\leq_p$-monotone and has a least fixpoint, called the *well-founded* fixpoint of A and denoted with $WF(A)$. Moreover, this fixpoint can be computed by transfinite iteration of S starting from the bottom element in the $\leq_p$ order which is $(\bot, \top)$.

The standard partial approximating operator of the T_P operator is Fitting's three-valued Φ_P operator [6]. The Kripke-Kleene fixpoint of Φ_P is equal to the Kripke-Kleene semantics of P [6]. Although partial stable models [14] are defined in a very different way they do coincide with partial stable fixpoints of Φ_P [3]. The definition of exact stable fixpoints is equivalent with the definition of stable models by Gelfond and Lifschitz [9]. Let $GL(P; I)$ denote the Gelfond-Lifschitz transformation of a program P with respect to an interpretation I. It is easy to show that $\Phi^1_P(I_1, I_2) = T_{GL(P;I_2)}(I_1)$. Thus $lfp(\Phi^1_P(\cdot, I))$ is equal to the least model of $GL(P; I)$. Consequently, I is an exact stable fixpoint of Φ_P if and only if I is a stable model of P. The computation of the well-founded fixpoint based on the partial stable operator of Φ_P is very similar to the bottom-up evaluation technique based on the doubled program by Kemp, Srivastava, and Stuckey [10].

3.2 Partial Stable Models of Aggregate Programs

We are now ready to define our semantics. Our goal is to define a partial approximating operator Φ^{aggr}_P of T^{aggr}_P which, for programs without aggregates, coincides with the Φ_P operator. The first step is to extend the evaluation function $[\![s]\!]_I$ of set expressions to partial interpretations as follows:

$$[\![s]\!]_{(I_1,I_2)} = (w(s^{I_1}), w(s^{I_2})).$$

The result of $[\![s]\!]_{(I_1,I_2)}$ is a partial multiset of the form (M_1, M_2), i.e. M_1 and M_2 are multisets such that $M_1 \subseteq M_2$. For example, consider the partial interpretation $(I_1, I_2) = (\emptyset, \{p, q\})$ and the set expression $s = \{p = 1, q = 1\}$. Then $[\![s]\!]_{(I_1,I_2)} = (\emptyset, M)$ where M is a multiset containing two times 1 and no other elements, i.e. $M(1) = 2$ and $M(x) = 0$ for all $x \neq 1$.

In three-valued logic, aggregates take partial multisets as input, so they have to be interpreted as partial relations. Instead of providing immediately a definition of partial aggregates we first give the basic properties which such partial aggregates have to satisfy such that the corresponding Φ^{aggr}_P operator will approximate T^{aggr}_P. Depending on the particular choice of partial aggregate relations we obtain different semantics.

Definition 5 (Approximating Aggregate). *Let* $\mathrm{R} \subseteq \mathcal{M}(D) \times D$ *be an aggregate relation. We say that the partial relation* $A_{\mathrm{R}} : \mathcal{M}(D)^c \times D \to \mathcal{THREE}$ *is an* approximating aggregate relation *of* R *if it satisfies the following properties:*

- A_{R} *extends* R*, i.e.* $A_{\mathrm{R}}((M, M), d) = (\mathrm{R}(M, d), \mathrm{R}(M, d))$*;*
- A_{R} *is* $\leq_p$*-monotone in the first argument, i.e. if* $(M_1, M_2) \leq_p (N_1, N_2)$ *then* $A_{\mathrm{R}}((M_1, M_2), d) \leq_p A_{\mathrm{R}}((N_1, N_2), d)$*.*

The next step is to define a valuation function for aggregate formulas in three-valued logic. For convenience, we view a partial interpretation (I_1, I_2) as a function $\tilde{I} : At \to \mathcal{THREE}$ defined as $\tilde{I}(a) = (I_1(a), I_2(a))$.

Definition 6 (Partial valuation function).

$$\begin{aligned}
\mathcal{H}_{\tilde{I}}(A) &= \tilde{I}(A), \textit{ for an atom } A \\
\mathcal{H}_{\tilde{I}}(\mathrm{R}(s,d)) &= A_{\mathrm{R}}([\![s]\!]_{\tilde{I}}, d), \textit{ for an aggregate atom } \mathrm{R}(s,d) \textit{ where } A_{\mathrm{R}} \\
&\quad \textit{is some approximating aggregate relation of } \mathrm{R} \\
\mathcal{H}_{\tilde{I}}(\neg F) &= \neg \mathcal{H}_{\tilde{I}}(F) \\
\mathcal{H}_{\tilde{I}}(F \wedge G) &= \mathcal{H}_{\tilde{I}}(F) \wedge_t \mathcal{H}_{\tilde{I}}(G) \\
\mathcal{H}_{\tilde{I}}(F \vee G) &= \mathcal{H}_{\tilde{I}}(F) \vee_t \mathcal{H}_{\tilde{I}}(G)
\end{aligned}$$

Based on $\mathcal{H}_{\tilde{I}}$ we define a three-valued immediate consequence operator Φ_P^{aggr} for aggregate programs.

Definition 7. $\Phi_P^{aggr}(\tilde{I}) = \tilde{J}$ *where* $\tilde{J}(A) = \bigvee_t \{\mathcal{H}_{\tilde{I}}(B) \mid A \leftarrow B \in P\}$.

For logic programs without aggregates the Φ_P^{aggr} operator coincides with the Fitting's Φ_P operator [6] and the Ψ_P operator defined by Przymusinski [14].

Theorem 1. Φ_P^{aggr} *is a partial approximating operator of* T_P^{aggr}.

Partial stable models of programs with aggregates are defined as the partial stable fixpoints of Φ_P^{aggr}.

We now give a concrete definition of the approximating aggregate relation A_{R} for every aggregate relation R. We use the idea of ultimate approximations [4]. In our previous work, we considered the ultimate approximating operator of the entire T_P operator [5] while here we consider the ultimate approximation only of aggregate relations.

Definition 8. *Let* $\mathrm{R} \subseteq \mathcal{M}(D) \times D$ *be an aggregate relation. The* ultimate approximating aggregate $U_{\mathrm{R}} : \mathcal{M}(D)^c \times D \to \mathcal{THREE}$ *of* R *is defined as follows:*

$$\begin{aligned}
((M_1, M_2), d) \in U^1_{\mathrm{R}} &\textit{ if and only if } \forall M \in [M_1, M_2] : \ (M, d) \in \mathrm{R} \\
((M_1, M_2), d) \in U^2_{\mathrm{R}} &\textit{ if and only if } \exists M \in [M_1, M_2] : \ (M, d) \in \mathrm{R}
\end{aligned}$$

Proposition 2. U_{R} *is a approximating aggregate relation of* R.

The ultimate approximating aggregate U_{R} is the most precise in the $\leq_p$-order among all possible approximating aggregate relations.

Proposition 3. $A_{\mathrm{R}}((M_1, M_2), d) \leq_p U_{\mathrm{R}}((M_1, M_2), d)$ *for every approximating aggregate relation* A_{R} *of* R*, partial multiset* (M_1, M_2)*, and element* $d \in D$.

For monotone aggregate relations the truth value can be computed directly on the boundary multisets.

Proposition 4. *Let* R *be a monotone aggregate relation. Then* $((M_1, M_2), d) \in U^1_{\mathrm{R}}$ *if and only if* $(M_1, d) \in$ R *and* $((M_1, M_2), d) \in U^2_{\mathrm{R}}$ *if and only if* $(M_2, d) \in$ R.

Example 2. Reconsider the program and the input of the party invitation problem from Example 1. In the well-founded model of the program, both a and b are undefined and there are no exact stable models. □

The semantics obtained by taking the ultimate approximating aggregate extends the semantics of definite aggregate programs given by the least fixpoint of the T_P^{aggr} operator.

Proposition 5. *A definite aggregate program P has a single partial stable model which is two-valued and is equal to $lfp(T_P^{aggr})$.*

4 Related Work

4.1 Ultimate Semantics of Aggregate Programs

In our earlier work on semantics of aggregates we studied the ultimate well-founded and ultimate stable models obtained from the ultimate partial approximating operator U_P^{aggr} [5]. Because the U_P^{aggr} is the most precise approximating operator of T_P^{aggr} the Φ_P^{aggr} operator is less precise than U_P^{aggr}.

Proposition 6. $\Phi_P^{aggr}(\tilde{I}) \leq_p U_P^{aggr}(\tilde{I})$ *for every partial interpretation* $\tilde{I}$.

More precise operators have more precise well-founded models (fewer undefined atoms) and a larger number of exact stable models [4]. So, we have the following relationship between the ultimate semantics of aggregate programs and the semantics which we define in this paper.

Proposition 7. $WF(\Phi_P^{aggr}) \leq_p WF(U_P^{aggr})$ *and* $ST(\Phi_P^{aggr}) \subseteq ST(U_P^{aggr})$.

For a large class of programs the two semantics will coincide. This is the case if, for example, a program has a two-valued well-founded model.

4.2 Stable Models of Weight Constraint Rules

The discussion of the related work in this and next subsection is with respect to the semantics obtained by interpreting aggregates with their ultimate approximations.

A closely related work is the stable model semantics of weight constraint rules defined by Simons, Niemelä, and Soininen [16] and implemented by the smodels system. A *weight constraint* is an expression of the form $l \leq s \leq u$ where s is a set expression and l, u are numbers. The intended interpretation of such expression is that the sum of the weights of the atoms satisfied by an interpretation has to be between the lower bound l and the upper bound u. The mapping between

programs with weight constraints and aggregate programs is given by translating a weight constraint $l \leq s \leq u$ to the conjunction $\text{SUM}_{\geq}(s, l) \wedge \text{SUM}_{\leq}(s, u)$.

Note that a weight constraint can also appear in a head of rule. Since our language does not allow aggregate atoms in heads of rules we study the relationship only for the class of programs which have simple atoms in the heads of rules. Under this restriction the two semantics coincide completely for programs which contain only weight constraints with lower bounds.

Proposition 8. *Stable models of a program with weight constraints without upper bounds coincide with exact stable models of aggregate programs.*

The next example demonstrates that for weight constraints with an upper bound the two semantics do not correspond in general.

Example 3. Consider the weight constraint program $P_{wc} = \{a \leftarrow \{not\ a = 1\} \leq 0\}$. Intuitively, the rule expresses the fact that a is true if *not* a is false or equivalently, a is true if a is true. The corresponding aggregate program $P = \{a \leftarrow \text{SUM}_{\leq}(\{not\ a = 1\}, 0)\}$ is definite according to Definition 2 and the T_P^{aggr} operator is monotone. Its least fixpoint is the empty set which is also equal to the well-founded and the single stable model. However, under the semantics of weight constraints a rule with an upper bound is translated to a rule with a lower bound by introducing an intermediate atom:

$$a \leftarrow not\ b.$$
$$b \leftarrow 1 \leq \{not\ a = 1\}.$$

This program is equivalent to the program $P' = \{a \leftarrow not\ b.\ b \leftarrow not\ a.\}$ which has two stable models $\{a\}$ and $\{b\}$. So the stable models of the original program are $\{a\}$ and $\emptyset$ which disagrees with the intuitive reading of the program. □

As the above example illustrates our semantics assigns a more intuitive meaning to programs with weight constraints with upper bounds. In particular, the exact stable models as defined in our approach are always minimal models of the program, which is not the case in the smodels approach. If under our semantics the transformation of weight constraints with upper bounds to weight constraints with lower bounds as used by the smodels system preserves the set of stable models then the two semantics are equivalent. The next proposition gives sufficient conditions for that.

Proposition 9. *If every negative weight literal in a weight constraint with upper bound does not depend on the atom in the head of the corresponding rule then the two semantics coincide.*

We now give an alternative definition of a reduct of weight constraints which makes the two semantics equivalent. A weight constraint of the form $l \leq s \leq u$ is first split in two constraints $l \leq s$ and $s \leq u$. The next step is to normalize the two weight constraints such that the first one contains only positive weights and the second one only negative weights. This is done using an algebraic transformation

given in [16]. By switching the signs of a weight literal and the associated weight and also updating the bounds we obtain an equivalent weight constraint. Let $sw_{L=w}$ be the transformation defined as follows:

$$sw_{L_i=w_i}(l \leq \{L_1 = w_1, \ldots, L_i = w_i, \ldots, L_n = w_n\} \leq u) = \\ l - w_i \leq \{L_1 = w_1, \ldots, \overline{L}_i = -w_i, \ldots, L_n = w_n\} \leq u - w_i.$$

For example $sw_{b=-2}(3 \leq \{a = 5, b = -2\} \leq 5) = 5 \leq \{a = 5, not\ b = 2\} \leq 7$.

The *reduct* C^I of a weight constraint C with lower bound with respect to an interpretation I is defined in exactly the same way as in [16]. The constraint C is first normalized to a constraint C^+ which contains only positive weights and has the form:

$$C^+ = l \leq \{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}, not\ b_1 = w_{b_1}, \ldots, not\ b_n = w_{b_n}\}.$$

The reduct C^I of C is defined as

$$C^I = l - \sum_{b_i \notin I} w_{b_i} \leq \{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}\}.$$

The *reduct* of a weight constraint C with upper bound is defined by first normalizing C to a constraint C^- which contains only negative weights and has the form:

$$C^- = \{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}, not\ b_1 = w_{b_1}, \ldots, not\ b_n = w_{b_n}\} \leq u.$$

The reduct C^I of C is defined as

$$C^I = \{a_1 = w_{a_1}, \ldots, a_m = w_{a_m}\} \leq u - \sum_{b_i \notin I} w_{b_i}.$$

The definition of a reduct of a program is the same as in [16].

Example 4. Reconsider the weight constraint program $P_{wc} = \{a \leftarrow \{not\ a = 1\} \leq 0.\}$ from Example 3. To obtain a reduct of the weight constraint $C = \{not\ a = 1\} \leq 0$ we first transform it to a normal form with only negative weights: $C^- = sw_{not\ a=1}(C) = \{a = -1\} \leq -1$. Because it does not contain negative literals $C^I = C^-$ for any interpretation I. So, the only stable model of the program P_{wc} is the least fixpoint of the program $\{a \leftarrow C^-.\}$ which is $\emptyset$. □

Proposition 10. *Stable models of a program with weight constraints using the definition of reduct above are equal with exact stable models of the corresponding aggregate program.*

4.3 Aggregate Atoms as Negative Literals

A line of research started by [11] and also used by the dlv system [2] and A-Prolog [8] defines a reduct of an aggregate program by treating aggregate atoms as negative literals. According to these definitions even definite aggregate programs can have non-minimal stable models [11].

Example 5. Consider the following aggregate program:

$$a \leftarrow \text{CARD}_{\geq}(\{a = 1\}, 1).$$

According to the stable semantics of [2,8,11] it has two stable models: $\emptyset$ and $\{a\}$. On the other hand under our semantics (as well as the semantics of weight constraints) it has a single stable model $\emptyset$. Note also that this is a definite aggregate program with monotone T_P^{aggr} operator with a least fixpoint $\emptyset$. □

We can establish a correspondence between the two semantics for aggregate programs which contain only negative aggregate atoms.

Proposition 11. *If an aggregate program P contains only negative aggregate atoms then exact stable models of P coincide with stable models of [2,8,11].*

5 Conclusions and Future Work

We introduced a framework for defining partial stable semantics of programs with arbitrary aggregate relations. By providing an appropriate definition of approximating aggregate one obtains semantics which extends the standard partial stable semantics for aggregate programs. Moreover, in contrast to some of the previous proposals of stable semantics of aggregate programs [2,8,11,16], the resulting semantics retains some of the important properties. In particular, two-valued stable models are always minimal models. We considered the semantics obtained by taking the most precise approximating aggregate relation — the ultimate approximating aggregate. This particular semantics has several advantages: it is the most precise and it is defined for arbitrary aggregate relations; for monotone aggregate relations it has a simpler and intuitive characterization.

In this paper we addressed only the first concern with the ultimate semantics of aggregate programs [5], namely the fact the it does not extend the standard partial stable semantics. The second concern was the higher computational complexity of the ultimate semantics. Because our semantics extends the standard semantics of logic programs it is interesting to see what impact adding aggregates has on the complexity. Our preliminary results indicate that even for aggregates which are polynomial time computable the complexity of the different semantics goes one level higher in the polynomial hierarchy. In particular, this is the case for the SUM and PROD aggregate functions. However, for certain aggregates, for example CARD, MIN, MAX, $\text{SUM}_{\leq}$, and $\text{SUM}_{\geq}$, we have shown that the complexity stays in the same class as for standard logic programs.

Our language does not allow aggregates in the heads of rules. We think that this is not a serious limitation because such programs can be translated to programs without aggregates in the heads by introducing additional atoms [12].

For simplicity of the presentation, we defined the semantics for a propositional language and set expressions of finite size. It can be easily extended to programs with variables by considering a suitable grounding of the program. The notions of aggregate relations and aggregate atoms remain exactly the same. We

only need to define set expressions with variables as done by other extensions of logic programs with aggregates [1,5,8,16]. It is possible that an intensionally defined multiset has an infinite size. Although many aggregate functions are not well defined for all infinite multisets, this is not a real problem because in our framework a partial aggregate function is represented as an aggregate relation.

References

1. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In *Proc. of the 18th Int. Joint Conference on Artificial Intelligence*, 2003.
2. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in DLV. In *Answer Set Programming: Advances in Theory and Implementation*, pages 274–288, 2003. online `CEUR-WS.org/Vol-78/`.
3. M. Denecker, V. Marek, and M. Truszczyński. Approximating operators, stable operators, well-founded fixpoints and applications in non-monotonic reasoning. In J. Minker, editor, *Logic-based Artificial Intelligence*, pages 127–144. 2000.
4. M. Denecker, V. Marek, and M. Truszczyńsky. Ultimate approximations in non-monotonic knowledge representation systems. In *Principles of Knowledge Representation and Reasoning*, pages 177–188. Morgan Kaufmann, 2002.
5. M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable model semantics for logic programs with aggregates. In P. Codognet, editor, *Int. Conf. on Logic Programming*, volume 2237 of *LNCS*, pages 212–226. Springer, 2001.
6. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
7. M. Fitting. Fixpoint semantics for logic programming a survey. *Theoretical Computer Science*, 278(1–2):25–51, 2002.
8. M. Gelfond. Representing knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408 of *LNCS*, pages 413–451. Springer, 2002.
9. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming, Proc. of the 5th International Conference and Symposium*, pages 1070–1080. MIT Press, 1988.
10. D. B. Kemp, D. Srivastava, and P. J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146(1–2):145–184, 1995.
11. D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *Proc. of the Int. Logic Programming Symposium*, pages 387–401. MIT Press, 1991.
12. V. Marek and J. Remmel. On logic programs with cardinality constraints. In *9th International Workshop on Non-Monotonic Reasoning*, pages 219–228, 2002.
13. I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *16th Int. Conf. on Very Large Data Bases*, pages 264–277, 1990.
14. T. Przymusinksi. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–464, 1990.
15. K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.
16. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
17. A. Van Gelder. The well-founded semantics of aggregation. In *11th ACM Symposium on Principles of Database Systems*, pages 127–138. ACM Press, 1992.

Improving the Model Generation/Checking Interplay to Enhance the Evaluation of Disjunctive Programs

Gerald Pfeifer*

Institut für Informationssysteme, TU Wien
Favoritenstraße 9-11, 1040 Wien, Austria
gerald@pfeifer.com

Abstract. Disjunctive Logic Programming (DLP) under the answer set semantics is an advanced formalism for knowledge representation and reasoning. It is generally considered more expressive than normal (disjunction-free) Logic Programming, whose expressiveness is limited to properties decidable in NP.
However, this higher expressiveness comes at a computational cost, and while there are several efficient systems for the normal case, we are only aware of few solid implementations for full DLP.
In this paper we investigate novel techniques to couple two main modules (a model generator and a model checker) commonly found in DLP systems more tightly. Instead of using the checker only as a boolean oracle, in our approach, upon a failed check it also returns a so-called unfounded set. Intuitively, this set explains why a model candidate is not an answer set, and the generator employs this knowledge to backtrack until that set is no longer unfounded, which is vastly more efficient than employing full-fledged model checks to control backtracking.
Furthermore, we invoke the checker not only for actual model candidates, but also on partial models during model generation to possibly prune the search space.
We implemented these approaches in DLV, a state-of-the-art implementation of DLP according to recent comparisons, and carried out experiments; tests on hard benchmark instances show a significant speedup of a factor of two and above.

1 Introduction

Disjunctive Logic Programming (DLP) without function symbols under the consistent answer set semantics [GL91] has slowly but steadily gained popularity since its inception in the early nineties of the previous century and now serves as an advanced formalism for knowledge representation and reasoning in areas such as planning, software configuration, model checking, and data integration [JNS+03,LPF+02].

For normal (non-disjunctive) logic programming under the answer set semantics, there are now a number of efficient implementations, notably Smodels [Sim96,SNS02], DLV [FP96,LPF+02], ASSAT [Zha02,LZ02], and Cmodels [Bab02], plus systems like aspps [ET01] or CCalc [McC99] that support closely related formalisms. In the disjunctive case, DLV and, more recently, GnT [JNSY00,JNS+03] have established themselves

* Supported by the European Commission under projects IST-2002-33570 INFOMIX, IST-2001-32429 ICONS, and IST-2001-37004 WASP.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 220–233, 2004.

(and it would be unfair to compare the performance of proof-of-concept research prototypes against these two); for details and benchmarks we refer to [JNS+03,LPF+02].

Originally, few applications really required the higher expressivity of DLP which allows to express every property of finite structures decidable in the complexity class Σ_2^P ($= \mathrm{NP}^{\mathrm{NP}}$, versus NP for normal logic programming, which means that under widely believed assumptions DLP is strictly more expressive [EGM97]). However, the use of DLP has been increasing: First, disjunction often allows for a more natural representation of problems not requiring the higher expressivity (and DLV, for example, detects such cases and avoids the overhead required for harder instances). And second, several applications from domains such as planning have been suggested (and implemented) [EFL+03,LRS01], which do require the full expressive power of DLP; an efficient implementation of the full language is paramount for these.

• DLP systems like DLV and GnT usually include two main modules: a model generator, which incrementally constructs model candidates using a backtracking procedure, and a model checker which verifies whether these candidates indeed are answer sets. In this paper, we provide a description of the intricate interaction of these two modules in DLV and introduce novel optimization techniques related to this interaction.

One optimization, implemented in GnT and DLV and first described in [JNSY00], is to perform partial model checks after a failed regular model check and backtrack until the (increasingly smaller) partial interpretation passes such a check (which means that the interpretation possibly can be extended to an answer set).

• We explore the possibility to use the model checker not just as a boolean oracle, but also let it return a so-called unfounded set in case the check fails. Intuitively, this set explains why the model candidate is not an answer set, and the generator can employ this knowledge during backtracking and avoid the more costly full partial model checks mentioned above in many cases.

• Furthermore, we consider another, orthogonal optimization to possibly prune the search space for model generation. There, we invoke the model checker not only when an actual model candidate has been found, but also on partial models during the process of model generation, and backtrack if that check fails.

• Finally, we implemented and refined these novel approaches in DLV, and carried out experiments. Results on 2QBF, a Σ_2^P-complete problem, show a reduction of execution time by more than half on average, and even more on particularly hard instances.

2 Disjunctive Logic Programming

In this section we briefly introduce (function-free) Disjunctive Logic Programming (DLP) under the consistent answer set semantics, provide a high-level overview of the implementation of DLV, and finally review previous results on model checking. For further background we refer to [GL91,EGM97,Bar02,LPF+02].

2.1 Syntax

A variable or constant is a *term*. An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms; for nullary predicates ($n = 0$) we

usually omit the parentheses. A *classical literal* is an atom a or a classically negated atom $\neg a$. A negation as failure literal (short *literal*) is either a positive literal c or a negative literal $\mathrm{not}\ c$, where c is a classical literal. A *(disjunctive) rule* r is

$$a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_k \wedge \mathrm{not}\ b_{k+1} \wedge \cdots \wedge \mathrm{not}\ b_m. \qquad n \geq 1,\ m \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals and r needs to be *safe*, i.e., each variable occurring in r must appear in one of the positive body literals $b_1, \cdots, b_k$ as well. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1 \wedge \ldots \wedge b_k \wedge \mathrm{not}\ b_{k+1} \wedge \ldots \wedge \mathrm{not}\ b_m$ is the *body* of r.

We denote by $H(r)$ the set $\{a_1, \ldots, a_n\}$ of the head literals, and by $B(r)$ the set $\{b_1, \ldots, b_k, \mathrm{not}\ b_{k+1}, \ldots, \mathrm{not}\ b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of classical literals occurring positively (resp., negatively) in $B(r)$: $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$.
Constraints are rules with an empty head ($n = 0$). A *program* is a finite set of rules (including constraints). A not-free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A literal, a rule, a program, etc. is *ground* if it does not contain any variables. Finite ground programs are also called *propositional*, and in the rest of this paper we will focus on such programs; the process of computing the ground equivalent of a program with variables has been the topic of separate research, e.g. [SNS02,Syr02,ELM+98].

2.2 Semantics

As usual, the *Herbrand Universe* $U_\mathcal{P}$ is the set of all constants appearing in a program $\mathcal{P}$, and the *Herbrand Literal Base* $B_\mathcal{P}$ is the set of all ground (classical) literals constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_\mathcal{P}$.

A *partial (or three-valued) interpretation* w.r.t. a program $\mathcal{P}$ is a pair $\langle T, F\rangle$ of subsets of $B_\mathcal{P}$; it is consistent if $T \cap F = \emptyset$ and there is no atom a s.t. $a \in T \wedge \neg a \in T$. For $x \in T$, $x \in F$, and $x \in B_\mathcal{P} - (T \cup F)$ we say x is true, false, and undefined, respectively. $I' = \langle T', F'\rangle$ is an *extension* of I (it extends I), if $T' \supseteq T$ and $F' \supseteq F$. A *(total) interpretation* I is a consistent partial interpretation where $T \cup F = B_\mathcal{P}$; it is often simply represented as $I = T$ (and F is implicitly given as $B_\mathcal{P} - I$). Total and partial models are defined as usual [Prz90,JNS+03].

For any rule r, the Ground Instantiation $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_\mathcal{P}$. For any program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set $\bigcup_{r\in\mathcal{P}} Ground(r)$.

Following [Lif96], we define the *Answer Sets* of a program $\mathcal{P}$ in two steps, using the ground instantiation $Ground(\mathcal{P})$: first we define the answer sets of positive programs; then we give a reduction of programs containing negation as failure to positive ones and use that to define answer sets of arbitrary programs.

Step 1: A total interpretation $I \subseteq B_\mathcal{P}$ is called *closed under a positive ground program* $\mathcal{P}$ if, for every $r \in \mathcal{P}$, $H(r) \cap I \neq \emptyset$ whenever $B(r) \subseteq I$, and $B(c) \not\subseteq I$ for every constraint $c \in \mathcal{P}$ (i.e., the constraint does not "trigger"). An interpretation I is an *answer set* for a positive ground program $\mathcal{P}$ if it is minimal (under set inclusion) among all total interpretations that are closed under $\mathcal{P}$.[1]

[1] Note that we only consider *consistent answer sets*, while in [GL91] also the inconsistent set of all possible literals can be a valid answer set.

Example 1. The positive program $\mathcal{P}_1 = \{a \vee \neg b \vee c.\}$ has the answer sets $\{a\}$, $\{\neg b\}$, and $\{c\}$. Its extension $\mathcal{P}_2 = \{a \vee \neg b \vee c.,\ \leftarrow a.\}$ has the answer sets $\{\neg b\}$ and $\{c\}$. Finally, $\mathcal{P}_3 = \mathcal{P}_2 \cup \{\neg b \leftarrow c.,\ c \leftarrow \neg b.\}$ has the single answer set $\{\neg b, c\}$. □

Step 2: The *reduct* or *Gelfond-Lifschitz transform* of a ground program $\mathcal{P}$ w.r.t. a set of literals $I \subseteq B_\mathcal{P}$ is the positive ground program $\mathcal{P}^I$, obtained from $\mathcal{P}$ by

1. deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap I \neq \emptyset$ holds;
2. deleting the negative body from the remaining rules.

An answer set of a (non-ground) program $\mathcal{P}$ is now a set $I \subseteq B_\mathcal{P}$ such that I is an answer set of $Ground(\mathcal{P})^I$.

Example 2. Given $\mathcal{P}_4 = \{a \vee \neg b \leftarrow c.,\ \neg b \leftarrow \text{not } a, \text{not } c.,\ a \vee c \leftarrow \text{not } \neg b.\}$ and $I = \{\neg b\}$, the reduct $\mathcal{P}_4^I$ is $\{a \vee \neg b \leftarrow c.,\ \neg b.\}$. I is an answer set of $\mathcal{P}_4^I$, and thus it is also an answer set of $\mathcal{P}_4$. Now consider $J = \{a\}$. The reduct $\mathcal{P}_4^J$ is $\{a \vee \neg b \leftarrow c.,\ a \vee c.\}$, and we can easily verify that J is an answer set of $\mathcal{P}_4^J$ and $\mathcal{P}_4$.

If, on the other hand, we take $K = \{c\}$, the reduct $\mathcal{P}_4^K$ is equal to $\mathcal{P}_4^J$, but K is not an answer set of $\mathcal{P}_4^K$: for $r = a \vee \neg b \leftarrow c$, the condition $B(r) \subseteq K$ holds, but $H(r) \cap K \neq \emptyset$ does not. Indeed, I and J are the only answer sets of $\mathcal{P}_4$. □

2.3 Answer Set Computation

In Figure 1 we provide a high-level description of the backtracking model generating procedure of the DLV system, which is similar to the one of GnT and the Davis-Putnam procedures commonly employed by SAT solvers [DP60]. Essentially, the model generator operates on a search tree where at every level we assume the truth or falsity of a classical literal and where, due to the use of heuristics, the order of literals on different branches may vary in general.

For simplicity, this description assumes that the program $\mathcal{P}$ as well as auxiliary data structures are globally accessible, and it omits the processes of parsing, computing a suitable ground version of the (possibly) non-ground input, and output.

```
function ModelGenerator(var I = ⟨T, F⟩ : ThreeValuedInterpretation) : bool;
begin
    if not DetCons(I) then return false;          (* inconsistency *)
    if T ∪ F = B_P then    (* no literal remains undefined in I *)
        return IsAnswerSet(I);
    Select an undefined literal A using some heuristics;
    if ModelGenerator(⟨T ∪ {A}, F⟩) then
        return true;
    else
        return ModelGenerator(⟨T, F ∪ {A}⟩);
end function ;
```

Fig. 1. Answer Set Computation

The computation is started by invoking ModelGenerator() with the empty three-valued interpretation $\langle\{\}, \{\}\rangle$, where every classical literal in $B_\mathcal{P}$ is set to undefined.[2] If

[2] In case of a three-valued interpretation every classical literal in $B_\mathcal{P}$ is either true, false, or undefined. If a literal assumes more than one of these, the interpretation is inconsistent.

$\mathcal{P}$ has an answer set which is a superset of I, ModelGenerator() returns true and sets I to this answer set; it returns false otherwise.

First, DetCons(I) adds to I literals which are true and false, resp., in all answer sets extending I (i.e., it adds deterministic consequences derivable from $\mathcal{P}$ and I, such as deriving the head of a positive normal rule r as true if $B^+(r) \subseteq I$). DetCons() returns false if these derivations result in inconsistency, in which case also ModelGenerator() backtracks and returns false. If no inconsistency occurred, and no literal in I is left undefined, we have a model candidate and invoke the model checker to determine whether it is indeed an answer set. Else we choose one of the undefined literals, assume it true and recurse; in case this does not lead to an answer set, we assume the complement of that literal true (that is, we assume the literal itself false) and recurse as well. This proceeds until we encounter an answer set or we have exhausted the entire search space.

We can easily see that there are three sources of complexity here in addition to the backtracking search itself: DetCons(), choosing an undefined literal, and the model check performed by IsAnswerSet().

By means of suitable data structures DLV performs DetCons() in linear time [DG84,CFLP02]. [SNS02] and [FLP01] provide more details on heuristics for the normal and disjunctive cases, respectively, and [KLP03] provides an in-depth description of the model checker of the DLV system. In Section 3 we will present optimizations to improve (partial) model checking and the pruning of the search tree.

2.4 Model Checking

In the following we review some previous results on model checking [KLP03,LRS97] and then proceed with improving upon the basic algorithm from Section 2.3.

The crucial concept for model checking in DLV is the notion of unfounded sets, which better lends itself for implementation than the original definition of answer sets.

Definition 1. (based on Definition 3.1 in [LRS97] and [KLP03]) Let I be a total interpretation for a program $\mathcal{P}$. A set $X \subseteq B_{\mathcal{P}}$ of ground classical literals is an *unfounded set* for $\mathcal{P}$ w.r.t. I if, for each rule $r \in Ground(\mathcal{P})$ such that $X \cap H(r) \neq \emptyset$, at least one of the following conditions holds:

C_1. $(B^+(r) \not\subseteq I) \vee (B^-(r) \cap I \neq \emptyset)$, that is, the body is false w.r.t. I.
C_2. $B^+(r) \cap X \neq \emptyset$, that is, some positive body literal belongs to X.
C_3. $(H(r) - X) \cap I \neq \emptyset$, that is, a literal in the head, distinct from the elements in X, is true w.r.t. I.

An interpretation I for a program $\mathcal{P}$ is called *unfounded-free* if and only if no non-empty subset of I is an unfounded set for $\mathcal{P}$ w.r.t. I. □

Intuitively, the presence of an unfounded set $X \subseteq I$ w.r.t. a model I of $\mathcal{P}$ indicates that I is not an answer set, because it is not minimal and some literals true in I can be falsified so that the resulting interpretation I' is still a model of $\mathcal{P}$. Formally, this can be stated as follows:

Proposition 1. *(Theorem 4.6 in [LRS97]) Let I be a model for a program $\mathcal{P}$. I is an answer set of $\mathcal{P}$ if and only if it is unfounded-free.* □

Example 3. Consider $\mathcal{P}_2$ from Example 1, which has three models: $\{\neg b\}$, $\{c\}$, and $\{\neg b, c\}$. The first two are trivially unfounded-free, for they do not have any non-empty proper subset and are not unfounded sets themselves, so both are in fact answer sets. $\{\neg b, c\}$ contains two unfounded sets, namely $\{\neg b\}$ and $\{c\}$, and is not an answer set. □

In general, checking whether a model I for $\mathcal{P}$ is an answer set is a co-NP-complete task, and DLV solves it by means of a translation of $\mathcal{P}$ and I to a satisfiability (SAT) problem which is unsatisfiable if and only if I is unfounded-free (and therefore an answer set). If we consider the resulting SAT instance as a functional problem its solutions are the unfounded sets for $\mathcal{P}$ w.r.t. I. For further details we refer to [KLP03].

Proposition 2. *(Partial Model Checking) Given a partial model I of a program $\mathcal{P}$, let $\mathcal{P}'$ be the program constructed by removing all literals which are undefined in I from $\mathcal{P}$, and let I' be the total interpretation obtained by keeping the positive (i.e., true) part of I and assuming all other literals false.*

If I' is not unfounded-free w.r.t. $\mathcal{P}'$, no total model that is an extension of I is unfounded-free w.r.t. $\mathcal{P}$. In other words, no extension of I is an answer set of $\mathcal{P}$. □

Proof. Let U be an unfounded set of $\mathcal{P}'$ w.r.t. I'. Furthermore let R and R' be the sets of all rules in $\mathcal{P}$ and $\mathcal{P}'$, respectively, that have one of the elements of U in the head.

It is sufficient to show that U remains an unfounded set however we assign the undefined literals in I the values true and false. There is a many-to-one correspondence between the rules in R and those in R' (different rules in R may be reduced to the very same rule in R'), so it is sufficient to show that if one of the conditions C_1–C_3 from Definition 1 holds for $r' \in R'$ w.r.t. I' it will also hold for all corresponding rules $r \in R$ w.r.t. each extension E of I:

(C_1) If the body of r' is false w.r.t. I' some literal $l \in B^+(r')$ is false w.r.t. I' or some literal $l \in B^-(r')$ is true w.r.t. I'. In either case, l is not undefined in I (or it would have been removed from r' by virtue of the construction of $\mathcal{P}'$) and thus will retain its value in every extension E of I, which in turn means that the body of every rule $r \in R$ corresponding to r' remains false w.r.t. E.

(C_2) If $l \in U$ is in $B^+(r')$, by the construction of r', l is also in $B^+(r)$ for every rule $r \in R$ corresponding to r'.

(C_3) Similarly, if there is a literal in the head of r' that is in I' but not in U, that literal is also true in I by the construction of I' from I and thus will retain its value in every extension E of I. □

3 Improving the Model Generation and Checking Interplay

Partial Model Checking during Backtracking. As mentioned before, both GnT and DLV implement an optimization first described in [JNSY00], where once a (total) model check fails, we backtrack and perform partial model checks during backtracking until we encounter a partial model (a three-valued interpretation) which passes such a partial check, or we reach the root of the search tree.

To that end, we introduce a global flag "state" which, when set to "check_failed", indicates that we are in this special backtracking mode. And we add a function IsAnswer-Set_Partial() which is similar to IsAnswerSet(), but ignores undefined literals occurring

```
var state : { normal, check_failed } := normal;
function ModelGenerator(var I = ⟨T, F⟩ : ThreeValuedInterpretation) : bool;
begin
    if not DetCons(I) then return false;          (* inconsistency *)
    if T ∪ F = B_P then    (* no literal remains undefined in I *)
        if not IsAnswerSet(I) then
            state:=check_failed; return false;
        else
            return true;
    Select an undefined literal A using some heuristics;
    if ModelGenerator(⟨T ∪ {A}, F⟩) then
        return true;
    else
        if state = check_failed then
            if not IsAnswerSet_Partial(I) then
                return false;
            else
                state:=normal;
        return ModelGenerator(⟨T, F ∪ {A}⟩);
end function ;
```

Fig. 2. Employing Partial Model Checking

in rules. That way it returns true if and only if I contains an unfounded set that will remain unfounded for *every possible totalization* of I (cf. Proposition 2) which allows us to continue backtracking, confident that there cannot be any solution left in that part of the search tree. The full, updated algorithm is depicted in Figure 2.

Quick Partial Model Checking. We improve upon this algorithm by further exploiting the results from Section 2.4 and a simple, but momentous, observation: when performing partial model checks during backtracking, the unfounded set determined by IsAnswerSet_Partial() will often be the same as the one originally found by IsAnswerSet().

Now we know that checking whether a set of classical literals is unfounded w.r.t. a program $\mathcal{P}$ and a (total or partial) interpretation I can be done in linear time.[3] So, instead of using the model checker only as a boolean oracle, whenever it encounters an unfounded set we also have it extract and return that set. For successive partial model checks we then first test whether the set is still unfounded w.r.t. I. If this is the case, we know that also a full partial model check of I would fail, avoid that costly check, and continue backtracking.

Otherwise, we need to bite the bullet and perform a full partial model check, as I may still contain an unfounded set different from the original one. Fortunately, IsUnfoundedSet() is extremely light and experiments confirmed that even in cases where this optimization does not succeed very often (or not at all) the overhead is hardly measur-

[3] This directly follows from Definition 1 under the assumption that checking whether a literal is contained in a set of literals can be done in O(1), as is the case for DLV and GnT which, similar to SAT solvers, internally represent ground literals as integers and interpretations (and unfounded sets in the case of DLV) as bitmaps.

```
var state : { normal, check_failed } := normal;
    ufset: SetOfClassicalLiterals;
function ModelGenerator(var I = ⟨T, F⟩ : ThreeValuedInterpretation) : bool;
begin
    if not DetCons(I) then return false;          (* inconsistency *)
    if T ∪ F = B_P then    (* no literal remains undefined in I *)
        if not IsAnswerSet(I,ufset) then
            state:=check_failed; return false;
        else
            return true;
    Select an undefined literal A using some heuristics;
    if ModelGenerator(⟨T ∪ {A}, F⟩) then
        return true;
    else
        if state = check_failed then
            var known_uffree : bool := false;
            if ufset ≠ ∅ then
                known_uffree := IsUnfoundedSet(ufset,I);
            if not known_uffree then
                known_uffree := not IsAnswerSet_Partial(I); ufset:= ∅;
            if known_uffree then
                return false;
            else
                state:=normal;
        return ModelGenerator(⟨T, F ∪ {A}⟩);
end function ;
```

Fig. 3. Employing Optimized Partial Model Checking

able. For normal and head-cycle free programs, the model generator of DLV completely avoids unfounded sets [CFLP02], and there is no overhead at all.

The full algorithm exploiting this new approach is displayed in Figure 3.

Increasing Opportunities for Quick Partial Checks. Finally, we further improve on this algorithm by also extending IsAnswerSet_Partial() as described above, and let it extract an unfounded set whenever it encounters an interpretation which fails the check (and thus cannot be extended into an answer set). That way, after one or more full partial model checks during backtracking, we may again switch into "quick mode" and a sequence of expensive full partial model checks can well be decomposed into alternating sequences of full and quick partial checks.

In terms of pseudo-code, we only need to replace the following line in Figure 3

known_uffree := **not** IsAnswerSet_Partial(I); $ufset$:= $\emptyset$;

by the following: known_uffree := **not** IsAnswerSet_Partial(I,$ufset$);

Partial Checks Forwards. Another novel approach is to speculatively perform partial model checks while moving forwards (as opposed to backtracking) in the search tree and there we obtained mixed results. "Brute force", where we perform such a check at every (or every n-th) level of the tree was a gain for many hard problems, but proved too

expensive in general, so after thorough experiments on various benchmarks (including all relevant problems used in [LPF+02]), we developed the following, cautious heuristics:

- Partial Model Checks Forwards are performed only for disjunctive programs,
- which are not head-cycle free (as for head-cycle free programs, the model generator of DLV already prevents the generation of unfounded sets [CFLP02]), and
- only if the previous model candidate failed to pass the final model check.

In that case we note the recursion level the failed total check occurred ($high$) and the level that partial model checks as described before lead to during backtracking (low, $low \leq high$) and perform a partial model check when reaching level

$$low + (high - low) * 1/3 \tag{1}$$

while descending the search tree again.

This check is based on the intuition that in the current part of the search tree levels low–$high$ carry some risk of introducing an unfounded set (as they did last time). We conducted a series of experiments using the instances described in the next section and $1/3$, $1/2$, and $2/3$ as factors in Equation 1 above, and while all of these brought noticeable speedups, the former was best overall.

4 Benchmarks

To assess the impact of the optimizations presented in Section 3, we use Quantified Boolean Formulas (2QBF), a well-known Σ_2^P-complete problem [Pap94] that already proved to be a suitable benchmark problem in other recent comparisons [KLP03, LPF+02].

Given a Quantified Boolean Formula $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a formula in 3DNF[4] over $X \cup Y$, the problem is to decide whether Φ is valid or not.

The transformation from 2QBF to disjunctive logic programming is a variant of a reduction used in [EG95], where we separate the actual problem instances and the following general encoding $\mathcal{P}_{2QBF}$:

$$\begin{aligned}
t(X) \vee f(X) &\leftarrow exists(X).\\
t(Y) \vee f(Y) &\leftarrow forall(Y).\\
w &\leftarrow conjunct(X, Y, Z, Na, Nb, Nc) \wedge\\
&\qquad t(X) \wedge t(Y) \wedge t(Z) \wedge f(Na) \wedge f(Nb) \wedge f(Nc).\\
t(X) &\leftarrow w \wedge forall(X).\\
f(X) &\leftarrow w \wedge forall(X).\\
&\leftarrow \text{not } w. \qquad t(true). \qquad f(false).
\end{aligned}$$

A concrete 2QBF instance Φ is then encoded by a set F_Φ of facts:

- $exists(v)$, for each existential variable $v \in X$;

[4] Disjunctive normal form with three propositional variables per clause.

- $forall(v)$, for each universal variable $v \in Y$; and
- $conjunct(p_1, p_2, p_3, q_1, q_2, q_3)$, for each disjunct $C_l = l_1 \wedge l_2 \wedge l_3$ in ϕ, where (i) if l_i is a positive atom v_i, then $p_i = v_i$, otherwise p_i= "$true$", and (ii) if l_i is a negated atom $\neg v_i$, then $q_i = v_i$, otherwise q_i="$false$".

For example, $conjunct(x_1, true, y_4, false, y_2, false)$, encodes $x_1 \wedge \neg y_2 \wedge y_4$.

Φ is valid, if and only if $\mathcal{P}_{2QBF} \cup F_\Phi$ has an answer set.

Benchmark Instances. We utilized benchmark instances from the extensive comparison of Answer Set Programming Systems in [LPF+02]. There, we randomly generated 50 instances per problem size such that the number of $\forall$-variables is equal to the number of $\exists$-variables (that is, $|X| = |Y|$), each conjunct contains at least two universal variables. and the number of clauses is equal to the number of variables (that is, $|X| + |Y|$). This schema was suggested by Gent and Walsh [GW99], and results there and in [LPF+02] confirm the hardness of the instances generated that way.

Compared Systems. We took the 2003-05-16 release of DLV and created four variants thereof: DLV$_{orig}$, which implements the strategy in Figure 2 and matches the behavior of the 2002-04-12 release of DLV; DLV$'$, with the algorithm from Figure 3; DLV$''$, which also updates the cached unfounded set during backtracking and is basically identical to the 2003-05-16 release; and DLV$_f''$, which cautiously performs partial model checks forwards as described above.

(A detailed comparison between DLV$''$ and GnT version 2, using a superset of the problems in this paper, can be found in [LPF+02].)

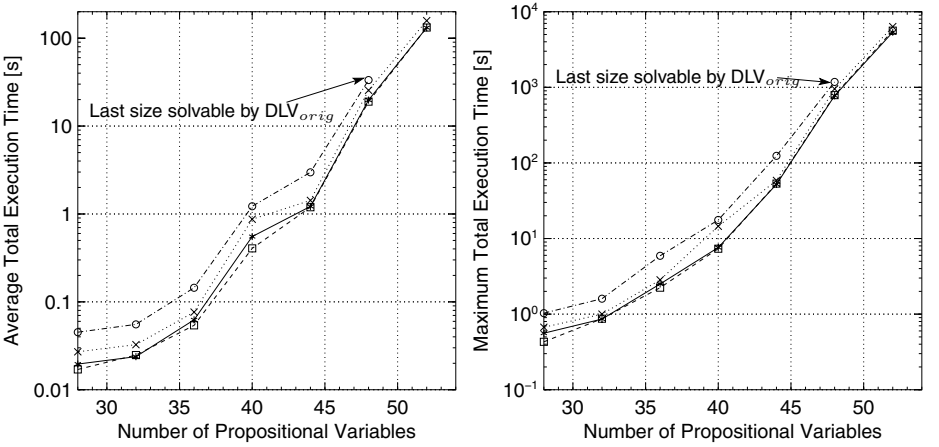


Fig. 4. Benchmark results for 2QBF

Environment and Execution. Benchmarks were performed on an Intel Pentium IV 2.4 GHz machine with 512 MB of memory, using FreeBSD 4.8 and GCC 2.95 with -O3 optimization to generate executables. We allowed a maximum running time of 7200 seconds (2 hours) per instance and a maximum memory usage of 256 MB.

Results. Cumulated results are provided in Figure 4. The graph to the left shows the average execution time for each system over the 50 instances per problem size; the

	Partial Checks		Quick Partial Checks				Overall Execution				
2QBF	Number		Number		Percentage		Time [s]				Speed
Instan.	–DLV″	DLV″$_f$	DLV′	DLV″	DLV′	DLV″	DLV$_{orig}$	DLV′	DLV″	DLV″$_f$	-up
40.6	27493	25801	15356	16821	55.9%	61.2%	16.5	8.8	7.6	7.3	*2.3*
40.28	33663	18558	2960	22000	8.8%	65.4%	17.6	14.5	7.6	4.3	*4.1*
40.29	18206	10794	4148	10955	22.8%	60.2%	9.9	7.2	4.4	2.7	*3.7*
40.39	28041	20807	5450	16140	19.4%	57.6%	16.1	12.2	7.5	5.4	*3.0*
44.22	201807	199988	132104	135038	65.5%	66.9%	124.1	58.6	52.1	53.3	*2.3*
44.42	19314	11706	14122	16444	73.1%	85.1%	10.7	4.2	3.1	2.3	*4.7*
48.10	1666943	1691654	210568	422252	12.6%	25.3%	1178.8	954.2	774.5	789.0	*1.5*
48.15	37743	19333	14888	28432	39.4%	75.3%	23.4	14.5	8.2	5.3	*4.4*
48.18	163636	139611	44120	85724	27.0%	52.4%	118.3	83.8	58.4	47.6	*2.5*
48.44	182439	112032	86715	117204	47.5%	64.2%	118.8	64.9	48.0	35.5	*3.3*
48.46	19020	10479	2986	12300	15.7%	64.7%	12.3	9.7	5.0	3.1	*4.0*
48.47	57605	54811	15424	26380	26.8%	45.8%	34.9	24.6	18.6	16.9	*2.1*
48.48	264435	170326	70390	164814	26.6%	62.3%	165.4	115.5	69.7	46.3	*3.6*
52.11	130943	75094	29376	85568	22.4%	65.3%	92.5	68.9	36.8	23.0	*4.0*
52.21	20695	9664	8542	17354	41.3%	83.9%	14.6	8.9	4.3	2.7	*5.4*
52.29	83294	41268	16062	56013	19.3%	67.2%	56.5	43.7	22.1	12.7	*4.4*
52.31	3353087	2401022	1935988	2093868	57.7%	62.4%	2222.8	1072.5	913.5	774.0	*2.9*
52.37	9800	5245	2180	6148	22.2%	62.7%	6.8	5.1	2.8	1.6	*4.2*
52.39	737280	518144	209700	384930	28.4%	52.2%	504.4	344.5	249.5	160.9	*3.1*
52.41	8539647	9225214	946566	1807494	11.1%	21.2%	–	6349.3	5355.8	5624.6	–
56.23	694837	365500	463000	578278	66.6%	83.2%	535.6	213.9	147.1	98.9	*5.4*
60.11	729792	508118	355689	461958	48.7%	63.3%	580.0	314.1	240.2	181.3	*3.2*
60.20	1989945	1023892	199920	1446084	10.0%	72.7%	1492.8	1212.8	517.1	290.0	*5.1*
60.21	18811	9412	9846	14561	52.3%	77.4%	14.2	7.2	4.6	3.4	*4.2*
60.31	40480	22912	14694	33721	36.3%	83.3%	30.1	19.2	8.6	5.5	*5.5*
60.33	241128	131444	67753	162331	28.1%	67.3%	194.2	134.0	74.1	51.9	*3.7*
60.36	50569	22880	7366	38958	14.6%	77.0%	40.7	32.0	13.1	7.7	*5.3*
64.31	56656	26400	21868	46030	38.6%	81.2%	43.5	27.0	13.0	7.7	*5.6*
64.33	637186	619266	312417	333073	49.0%	52.3%	543.2	307.4	270.7	259.0	*2.1*
64.34	24475	15819	6176	18408	25.2%	75.2%	21.2	15.1	6.7	4.8	*4.4*
64.16	2153919	984889	505896	1490504	23.5%	69.2%	1808.0	1307.0	669.5	382.7	*4.7*
64.6	3156869	2141079	908358	1901420	28.8%	60.2%	3252.6	2213.9	1402.0	1028.4	*3.2*
64.7	723407	344002	56068	566996	7.8%	78.4%	755.2	629.2	215.4	135.3	*5.6*
76.39	226053	151372	32172	159434	14.2%	70.5%	249.9	198.5	83.6	57.5	*4.3*

Fig. 5. Detailed benchmarks results for hardest 2QBF instances

graph to the right shows the maximum time taken. The plot for a system stops whenever that system failed to solve some problem instance within the given time/memory limits.

To study the performance of our optimizations in more detail, we extracted the 34 hardest instances (that still could be solved) from our testbed. This includes several instances of larger sizes than were relevant for the full tests, where a system was killed once it encountered the first untractable instance. (Informal tests confirmed the hardness of these instances, in that GnT was not able to solve many of them within two hours; we again refer to [LPF$^+$02] for a more complete comparison.)

Table 5 shows detailed results for these 34 instances, specifically, from left to right: the overall number of partial model checks for DLV$_{orig}$, DLV′, and DLV″ (where it is always

the same) on the one hand, and DLV_f'' on the other; the number as well as percentage of those instances where the quick partial model check using IsUnfoundedSet() was sufficient and we could avoid the more expensive full check, both for DLV′ and DLV″ (for DLV_{orig} these are always zero by definition); the overall execution times for DLV_{orig}, DLV′, DLV″, and DLV_f''; and, finally, the speedup from DLV_{orig} to DLV_f''.

5 Conclusions

The benchmarks show that the optimizations we derived and implemented are a clear win already on the averages over the 50 2QBF instances per size, where we observe a speedup of about 2 (specifically, 2.6, 3.0, 2.5, and 1.8 for sizes 36, 40, 44, and 48, respectively). And, unlike DLV_{orig}, the optimized variants of DLV are able to solve all instances of size 52.

Results are even more favorable on the hardest instances of 2QBF where a significant amount of time is spent on (partial) model checking. Already DLV′ is a measurable improvement, but for DLV″, on average more than 50% of the partial checks enjoy the superior performance of our novel approach, resulting in speedups of about 2–3 in most cases. Furthermore, for these instances, DLV_f'' is able to shave even more off the run-time, resulting in overall speedups of a factor of 2–5.

These results are encouraging and we are working to further improve the interplay of model generator and checker, where we believe that better and more aggressive heuristics concerning the application of partial model checks forwards will prove crucial.

We note that these optimizations are applicable to other (DLP) systems as well. In particular, partial model checks forwards can be used whenever it is possible to "totalize" models during generation. And in case of systems like GnT that use counter-models (i.e., subsets of the original model which are still models) instead of unfounded sets, the reuse of the latter can be simulated by caching a counter-model and subsequently, during backtracking, checking whether a partial model at hand is a superset of the cached model. In general, however, using unfounded sets seems more efficient, as these tend to be (much) smaller than counter-models. Also, they focus on the critical components of (non-minimal) model candidates and often are more "persistant" during backtracking.

In fact, the structure of GnT is quite similar to the one assumed in this paper, and it may be worthwhile to port the model checking techniques employed by DLV to GnT. (Analyses and detailed benchmarks have shown that moving in the other direction, and port the model checking techniques from GnT to DLV, would be a loss. We believe this to be largely due to the efficiency of the SAT-based approach used by DLV.)

Acknowledgments. I am very grateful to Nicola Leone for fruitful discussions and suggestions related to this work as well as our long running cooperation in general.

Also, I would like to thank Axel Polleres for carefully proof-reading a draft of this paper and the other members of the DLV team whose continuous support provided the foundation to base this work on, especially Wolfgang Faber and Simona Perri as well as Tina Dell'Armi, Giuseppe Ielpa and Francesco Calimeri for their work on the core system and Christoph Koch.

And thanks to the anonymous reviewers for very valuable & constructive comments.

References

[Bab02] Y. Babovich. Cmodels homepage. http://www.cs.utexas.edu/users/tag/cmodels.html.

[Bar02] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.

[CFLP02] F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning Operators for Answer Set Programming Systems. In *NMR'2002*, pp. 200–209, April 2002.

[DG84] W. F. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfability of Propositional Horn Formulae. *JLP*, 3:267–284, 1984.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *JACM*, 7:201–215, 1960.

[EFL+03] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. To appear in ACM TOCL, 2003.

[EG95] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *AMAI*, 15(3/4):289–323, 1995.

[EGM97] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM TODS*, 22(3):364–418, September 1997.

[ELM+98] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. Progress Report on the Disjunctive Deductive Database System `dlv`. *FQAS'98*, pp. 148–163. Springer.

[ET01] D. East and M. Truszczyński. aspps – An Implementation of Answer-Set Programming with Propositional Schemata. *LPNMR'01*, pp. 402–405. Springer.

[FLP01] W. Faber, N. Leone, and G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *IJCAI 2001*, pp. 635–640. Morgan Kaufmann.

[FP96] W. Faber and G. Pfeifer. `DLV` homepage, since 1996. http://www.dlvsystem.com/.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[GW99] Ian Gent and Toby Walsh. The QSAT Phase Transition. In *AAAI*, 1999.

[JNS+03] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. Tech. Report cs.AI/0303009, arXiv.org.

[JNSY00] T. Janhunen, I. Niemelä, P. Simons, and Jia-Huai You. Partiality and Disjunctions in Stable Model Semantics. *KR 2000, April 12-15*, pp. 411–419. Morgan Kaufmann.

[KLP03] C. Koch, N. Leone, and G. Pfeifer. Using SAT Checkers for Disjunctive Logic Programming Systems. *Artificial Intelligence*, 2003. To appear.

[Lif96] V. Lifschitz. Foundations of Logic Programming. *Principles of Knowledge Representation*, pp. 69–127. CSLI Publications, Stanford, 1996.

[LPF+02] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. Tech. Report cs.AI/0211004, arXiv.org, November 2002. Submitted to ACM TOCL.

[LRS97] N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112.

[LRS01] N. Leone, R. Rosati, and F. Scarcello. Enhancing Answer Set Planning. *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pp. 33–42.

[LZ02] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *AAAI-2002*, AAAI Press / MIT Press.

[McC99] N. McCain. The Causal Calculator, 1999. http://www.cs.utexas.edu/users/tag/cc/.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Prz90] Teodor C. Przymusinski. Well-founded Semantics Coincides with Three-valued Stable Semantics. *Fundamenta Informaticae*, 13:445–464, 1990.

[Sim96] P. Simons. Smodels Homepage, since 1996. http://www.tcs.hut.fi/Software/smodels/.

[SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[Syr02] T. Syrjänen. Lparse User's Manual, 2002. http://www.tcs.hut.fi/Software/smodels/.

[Zha02] Y. Zhao. ASSAT homepage. http://assat.cs.ust.hk/.

Using Criticalities as a Heuristic for Answer Set Programming

Orkunt Sabuncu[1], Ferda N. Alpaslan[1], and Varol Akman[2]

[1] Department of Computer Engineering
Middle East Technical University,
06531 Ankara, Turkey
{orkunt,alpaslan}@ceng.metu.edu.tr
[2] Department of Computer Engineering
Bilkent University,
06800 Ankara, Turkey
akman@cs.bilkent.edu.tr

Abstract. Answer Set Programming is a new paradigm based on logic programming. The main component of answer set programming is a system that finds the answer sets of logic programs. During the computation of an answer set, systems are faced with choice points where they have to select a literal and assign it a truth value. Generally, systems utilize some heuristics to choose new literals at the choice points. The heuristic used is one of the key factors for the performance of the system.
A new heuristic for answer set programming has been developed. This heuristic is inspired by hierarchical planning. The notion of criticality, which was introduced for generating abstraction hierarchies in hierarchical planning, is used in this heuristic. The resulting system (CSMODELS) uses this new heuristic in a static way. CSMODELS is based on the system SMODELS. The experimental results show that this new heuristic is promising for answer set programming. A comparison of search times with SMODELS demonstrate CSMODELS' usefulness.

1 Introduction

Answer Set Programming is a new programming paradigm. It is based on logic programming but solutions of a problem are not extracted from a proof session [1]. It is a model-theoretic approach. One writes a logic program for the problem at hand in such a way that the intended models of the program correspond to the solutions of the problem. The semantics used for selecting the intended models in answer set programming is *stable model semantics*. The models found by the semantics are called *stable models* or *answer sets* and each of the models correspond to a solution of the problem.

The main component of answer set programming is a system that finds the answer sets. Such systems can be seen as the implementations of stable model semantics.

The main algorithm for finding an answer set is common to almost all systems. The idea is a simple generate-and-test cycle [2]. In the generation phase,

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 234–246, 2004.

a candidate model is constructed for the input logic program. This model is a partial model. The aim is to transform this candidate model into an answer set. The system periodically checks whether the model at hand is an answer set while augmenting it in the test phase.

At *choice points* (see Sect. 3) the system chooses an uncovered atom and assigns it *true* or *false* as an interpretation to augment the candidate model. Some choices cause the system find an answer set very quickly, but some cause it to enter an incorrect search path and consume lots of time before it backtracks. So, heuristics are usually used for choosing an uncovered atom. They greatly affect the performance of the system.

Working on new heuristics is important for developing better systems for answer set programming [2]. In this work, a new system called CSMODELS[1] (Criticality SMODELS), which is based on SMODELS [3], is developed. CSMODELS uses a new heuristic whose foundation is *criticality*. The notion of criticality has been used in hierarchical planning [4].

Hierarchical planning is a way to deal with search space complexity of planning problems. Hierarchical planners attack the problem at different levels of detail. An *abstraction hierarchy* is used to define these levels.

There are successful abstraction hierarchies that improve the performance of the hierarchical planner, but there are also poor ones which cause considerable backtracking between levels. In a good abstraction hierarchy, the upper levels (the most abstract ones) should deal with the hardest part of the planning problem so the planner tries to solve them first. This property will limit the amount of backtracking between the levels during the plan finding process [5]. The notion of criticality was introduced in [4] for automatically generating abstraction hierarchies based on this property. Criticality of a literal in a planning problem is a numerical value and approximates the difficulty of finding a plan that achieves this literal.

There is also the difficulty of finding a literal in an answer set. Selecting the 'hardest' literals at the first choice points can limit backtracking and can help the system find an answer set quickly as in the case of hierarchical planning. This is the main motivation for our work. In CSMODELS, criticalities of the literals of a logic program are calculated to approximate the difficulty of finding them in an answer set. Then, these values are used as a heuristic at the choice points. It is not trivial to calculate the criticalities this time because the original method was for planning problems, not for logic programs. A method for applying criticality calculation for answer set programming is also developed.

The experimental results obtained with CSMODELS are encouraging. Generally, CSMODELS finds an answer set of a program more efficiently than SMODELS. This increase in the performance of search time is significant for especially large problems.

Background information about criticalities and how to calculate them are given in the next section. Section 3 describes the main algorithm of SMODELS. The essential contribution of this work, which is applying the notion of criti-

[1] http://www.ceng.metu.edu.tr/~orkunt/csmodels/ .

cality to answer set programming, is presented in Section 4. Section 5 describes CSMODELS. Section 6 includes the experimental results. In the final section, conclusions can be found.

2 Criticalities

Criticalities are used for generating abstraction hierarchies for planners. Planners use abstraction to reduce the complexity of the planning problem [6,7]. ABSTRIPS, ABTWEAK, ALPINE, and RESISTOR are some hierarchical planners [6].

Hierarchical planners use an abstraction hierarchy for the planning problem to generate and solve subproblems. An *abstraction hierarchy* divides the whole problem into pieces. A hierarchy level indicates which parts of the problem should be neglected or taken into account to form a subproblem corresponding to that level. After finding an abstract plan, the next step is to go one level below and find a plan for that level. The information that is neglected in the higher level is considered now as part of the problem. This is called *refinement*. This process continues level by level until the ground level (i.e., the lowest level) is reached.

Results of using abstraction in planning are generally encouraging. In some problems it can lead to an exponential reduction in the search space, improving the efficiency of plan finding [8].

There are also discouraging results [9,10]. The main cause of inefficiency is *trashing* [5] between the levels of abstraction hierarchy. During refinement, if no plan can be found in a level, backtracking to a more abstract level (the upper one) to search for another abstract plan is inevitable. There may be many possible abstract plans which cannot be refined. This will cause numerous backtrackings, leading to trashing. A good abstraction hierarchy should avoid trashing. To limit trashing, a system should try to solve a subproblem which constitutes the hardest part of the original problem first [5].

Bundy, Giunchiglia, Sebastini, and Walsh [4] provide a method for generating good hierarchies and provide details of an implementation called RESISTOR. RESISTOR sorts the precondition literals of a planning problem according to their difficulties to achieve (i.e., their costs). So, it partitions the problem into parts in terms of difficulty to generate a good abstraction hierarchy.

The method discussed in [4] uses a numerical simulation of the plan finding process. This method has introduced the notion of *criticality*. Criticality captures the cost of a literal; criticality of a literal is an approximation of the cost of achieving it. Criticalities have numerical values; smaller values correspond to easier-to-achieve literals, larger values correspond to harder-to-achieve literals.

The *criticality function* $C(p, n)$ gives the criticality value of literal p. There is an interpretation of $C(p, n)$ as *the difficulty of finding a plan of length* $\leq n$ *achieving* p. This interpretation gives rise to a group of criticality function definitions.[2] Our work is based on RESISTOR's criticality functions. After defining

[2] Other definitions and detailed information can be found in [4].

the criticality functions, criticality values are calculated in an iterative way. The iteration is on n starting from 0. So, we basically see $C(p, n)$ as a function that gives the criticality value of literal p at the n-th iteration. Note that the variable n refers to plan length in the interpretation just to find criticality function definitions.

RESISTOR's definition of $C(p, n)$ is inspired by electrical resistors. In calculating the difficulty of achieving p, operators whose effect is p can be regarded as electrical resistors connected in parallel. Fig. 1 shows this circuit. The more operators there are, the more paths there will be for achieving p. Consequently, finding p becomes less difficult. As in equation (1), $C(p, n)$ is calculated by the parallel sum (total resistance of parallel connected resistors) of the criticalities of operators whose effect is p and the initial criticality $C(p, 0)$ (the difficulty of finding a plan of length 0).

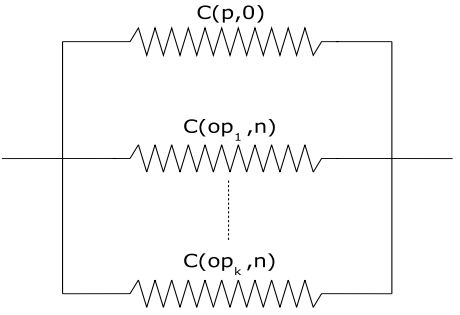


Fig. 1. Criticality circuit for the precondition p

$$\frac{1}{C(p,n)} = \frac{1}{C(p,0)} + \frac{1}{C(op_1,n)} + \ldots + \frac{1}{C(op_k,n)} \tag{1}$$

Equation (1) introduces *the notion of criticality of an operator*. $C(op, n)$ for an operator op is defined to make calculations simple. It is interpreted as the *difficulty of finding a plan of length 1 to n which ends with the occurrence of operator op*. For calculating the criticality of an operator, preconditions of that operator can be regarded as electrical resistors connected in series (Fig. 2).

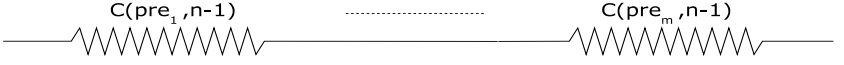


Fig. 2. Criticality circuit for the operator op

$$C(op, n) = C(pre_1, n-1) + \ldots + C(pre_m, n-1) \tag{2}$$

Since the interpretation of $C(op, n)$ expects the occurrence of *op* at the n-th step, all the preconditions must be achieved up to that point. This is why criticalities of preconditions at the $(n-1)$-th iteration are used in equation (2).

Using equations (1) and (2), criticalities of all the precondition literals are calculated iteratively. Initial criticality values for every literal (i.e., $C(p, 0)$ terms) are assigned to 1. Since criticality functions are monotonically decreasing and convergent [4], the limiting values of criticalities will lie in the interval [0,1] after some iterations.[3] RESISTOR continues to iterate until no change occurs in the values (within some predefined *accuracy*). This is a computationally practical way of terminating iterations.

Abstraction hierarchies are generated by sorting the final criticality values. However, what is important for our work is just how they are calculated.

3 SMODELS: An Answer Set Programming System

SMODELS has been developed at Helsinki University of Technology, and is now one of the most popular answer set programming systems. The system implements stable model semantics [11] for logic programs. Reference [3] explains SMODELS' philosophy and implementation in detail.

Input to SMODELS is variable free logic programs. Programs with variables are transformed to ground logic programs by a front-end system called LPARSE[4] [12]. SMODELS uses a candidate model which is an empty set at the beginning. It tries to augment this candidate model by adding literals deterministically according to the program by using the properties of stable model semantics. This process of generating new literals deterministically is equivalent to *expanding* the model at hand.

Expanding a model can cause situations in which both an atom and its negation are in the model at the same time. This contradicting situation is called a *conflict*.

At an intermediate stage of the search process, an atom can take one of three different possible values [2]: *true*, *false*, or *undefined*. If an atom has a value of undefined, it is an *uncovered* atom.

At the end of first expansion the candidate model corresponds to a *well-founded model* [13] of the program. The aim is to expand the candidate model until it becomes a stable model. The decision criterion for a candidate model to be a stable model is that all the atoms of the program should be covered by it and there should be no conflicts. At several stages during the whole search process there are uncovered atoms and SMODELS cannot deterministically assign values to these atoms (i.e., true or false). These stages are called *choice points*. At choice points, SMODELS should just select one of the uncovered atoms and give an interpretation as true or false. Then, the main algorithm will again try to expand the newly generated candidate model with the addition of the chosen

[3] Actually, if the initial criticality values are 1, then all the criticalities will be in this interval.

[4] Available at http://www.tcs.hut.fi/Software/smodels/ .

atom. Remember that there can be conflicts after expansion. An incorrect choice at a previous choice point can be the reason for a conflict. So, in case of a conflict SMODELS backtracks to the previous choice. Fig. 3 shows a simplified version of SMODELS' main algorithm [3].

```
function smodels(CM)   {CM: Candidate Model}
  CM := expand(CM)
  if conflict(CM) then
       return false
  else if no atom is undefined in CM then
       return true   {CM is a stable model}
  else
       c := Choose an uncovered atom
       if smodels(CM ∪ {c}) then
            return true
       else
            return smodels(CM ∪ {not c})
       end if
  end if
```

Fig. 3. Main algorithm of SMODELS

SMODELS' heuristic to choose an atom is based on minimizing the remaining search space after a choice is made [3]. SMODELS selects every uncovered literal temporarily and adds it to the candidate model at a choice point. It does so in order to determine what happens if it chooses that uncovered literal actually. After expanding the candidate model, there will possibly be newly derived literals. The number of these new literals is the heuristic score of the chosen literal. If the chosen literal is positive then the score of the literal is called *positive score* of that atom. Similarly, the score of a negative literal of the same atom is *negative score* of that atom.

Given the minimum of positive and negative scores of each atom, SMODELS' heuristic chooses the maximum one to guarantee that it reduces the search space maximally.

The heuristic of SMODELS is a *dynamic heuristic* [2]: at every choice point it recomputes the scores.

4 Applying the Criticality Notion for Answer Set Programming

Criticalities make it possible to find hard-to-achieve literals and to work on them first during hierarchical plan finding. This will reduce the amount backtracking between the levels of abstraction hierarchy.

The same idea can also be applied to answer set programming. During the answer set finding process, choosing the hard-to-find literals at the early choice

points can reduce the backtracking and permits the system to find an answer set quickly. There are costs of finding literals in an answer set similar to the costs of achieving literals in a planning problem. Knowing the costs of literals of an input logic program can be useful for a system at the choice points. This intuition is the main motivation for the heuristic developed in our work.

Every atom in the stable model has to be *grounded*. If an atom has at least one rule that has generated it, then it is grounded. Rules that have an atom in the head are possible *generators* of it.

The only condition for a rule to generate the literal in its head is that its body must be true. So, if we accept the rule's body as a *precondition* we can transform a rule into a *planning operator (action)*.[5] The *effect* of this planning operator is the head atom. Let all the generator rules for literal a below constitute a portion of a logic program.

$$a \leftarrow b,\ not\ c \quad (rule_1) \tag{3}$$

$$a \leftarrow d \quad (rule_2) \tag{4}$$

The corresponding planning operator for rule (3) is given below.

$$a \leftarrow^{r_1} b,\ not\ c \quad \Longrightarrow \quad \begin{array}{l} \text{Operator : } r_1 \\ \text{Preconditions : } b,\ not\ c \\ \text{Effect : } a \end{array}$$

When we consider the above rules as planning actions, equations (5–7) are used for calculating the criticality of literal a. Note that the initial criticality of a, $C(a, 0)$, is set to 1 like all the other literals' initial criticalities.

$$C(rule_1, n) = C(b, n-1) + C(not\ c, n-1) \tag{5}$$

$$C(rule_2, n) = C(d, n-1) \tag{6}$$

$$\frac{1}{C(a,n)} = \frac{1}{C(a,0)} + \frac{1}{C(rule_1,n)} + \frac{1}{C(rule_2,n)} \tag{7}$$

There is a term representing criticality value of a negative literal in equation (5). How can the criticality of a negative literal be found? There are no rules that can generate a negative literal in the sense of generating positive ones. In fact *negation-as-failure* means that if it is not possible to achieve an atom a, then *not* a is assumed to be true. Considering the properties of stable model semantics, we can define the necessary conditions for *not* a to be in the stable model as no generator rule will have a chance to generate a. The bodies of all generator rules must be interpreted as false, so literal *not* a is interpreted as true.

[5] Lin and Reiter [14] define logic rules as planning actions in the context of defining semantics for logic programs using situation calculus.

Here is the only rule generating *not a* formed by taking all the generator rules (3–4) of a into consideration:[6]

$$not\ a \leftarrow (not\ b \vee c),\ not\ d \quad (rule_{not\ a}) \tag{8}$$

Based on rule (8), the criticality of literal *not a* is calculated using the following equations:

$$C(rule_{not\ a}, n) = C((not\ b \vee c), n-1) + C(not\ d, n-1) \tag{9}$$

$$\frac{1}{C(not\ a, n)} = \frac{1}{C(not\ a, 0)} + \frac{1}{C(rule_{not\ a}, n)} \tag{10}$$

Calculating the criticality value of a negative literal is no different than calculating a positive literal except for compound literals with disjunction. Achieving a compound literal as a whole should be easier than achieving each one of the basic literals individually. So, the criticality value of a compound literal should be less than each of the criticalities of its basic literals. Calculating the criticality of compound literal $(notb \vee c)$ by the equation $C((notb \vee c), n) = C(notb, n) \times C(c, n)$ is a meaningful approximation, since the multiplication of two or more real numbers in the interval [0,1] is always smaller than each number (or equal to the smallest one at least).

A *constraint rule* is a rule that has no atom in the head (i.e., a rule that has an implicit false in the head). They are not used in criticality calculations, since they cannot be generators for any atom.

Using the above equations, we can calculate the criticalities of all the literals of a logic program if all the rules in the program are normal. But SMODELS enlarges the syntax and semantics of logic programs by extended rules [3]. These extended rules are *choice, cardinality,* and *weight* rules. Our system, CSMODELS, supports choice rules, but not cardinality or weight rules.

$$\{a\} \leftarrow b,\ not\ c \tag{11}$$

Rule (11) is a choice rule. A choice rule's head may not be true although its body is satisfied by the model unlike normal rules. One can rewrite the choice rule using only normal rules. But this translation introduces new atoms to the input program [3]. Choice rule (11) can be rewritten as two rules:

$$\begin{array}{l} a \leftarrow not\ c\_a,\ b,\ not\ c \\ c\_a \leftarrow not\ a,\ b,\ not\ c \end{array} \tag{12}$$

The atom *c_a* is a newly introduced atom. By treating choice rules as if two normal rules of the form (12), we can handle choice rules for criticality calculations. In this way we do not enlarge the program. Answer sets in the output do not contain the introduced atoms, since they are only used for criticality calculations.

[6] Body of the rule (8) is the inverse of the completion of literal a [15]. We can see the inverse of the completion of literal x as a generator rule for literal *not x*.

5 CSMODELS (Criticality SMODELS)

CSMODELS uses criticalities as a heuristic and is based on the SMODELS system. It implements the method of applying criticality calculation to logic programs of SMODELS described in Sect. 4. Implementation details of CSMODELS can be found in [16]. The main algorithm is the same as that of SMODELS except the parts related to the choice points.

CSMODELS calculates the criticality values at the first choice point. Unlike SMODELS, CSMODELS' heuristic is not dynamic. So, the same heuristic scores calculated at the first choice point are used at all the other choice points. At the first choice point, the candidate model at hand corresponds to a well-founded model (WFM) of the program. WFM covers some of the atoms of the input program, so there is no need to calculate the criticality values of the already covered atoms. We set the criticality of a literal in WFM to 0, indicating that it is very easy to achieve that literal or the literal is already achieved. Also, we set the criticality of the inverse of that literal to 1 indicating the impossibility or difficulty of achieving the literal.[7]

$$\text{If literal } l \in \text{WFM} \quad \Longrightarrow \quad \begin{array}{l} C(l) = 0 \\ C(not\, l) = 1 \end{array}$$

After calculating criticality values of all uncovered literals, CSMODELS finds the *criticality heuristic scores* to select a literal at the choice points. It temporarily selects every uncovered literal one by one and adds them to the candidate model just like SMODELS computes heuristic scores. Let x be an uncovered literal by the WFM, the expansion of the model formed by adding x to WFM will probably generate new literals that have been uncovered previously. Let these literals be

$$NewLiterals(x) = \{a,\, b, \ldots,\, not\, k,\, not\, l, \ldots\}. \tag{13}$$

CSMODELS finds the criticality heuristic score of literal x by the following equation:

$$\begin{array}{rl} Score(x) = & C(a) + (1 - C(not\, a)) + \\ & C(b) + (1 - C(not\, b)) \; + \ldots + \\ & C(not\, k) + (1 - C(k)) + \\ & C(not\, l) + (1 - C(l)) \; + \ldots \end{array} \tag{14}$$

The purpose of the score of a literal is to approximate the *difficulty of choosing that literal at a choice point*. Knowing that choosing x generates literal a, the system faces the cost of achieving a and avoiding the generation of *not* a when choosing x. That is why the terms $C(a)$ and $1 - C(not\, a)$ are added to the score. This is done for all the literals in $NewLiterals(x)$. Just like SMODELS, there are positive and negative criticality heuristic scores of an atom.

[7] Note that 1 is the maximum criticality that a literal can have.

The set *NewLiterals* for an uncovered literal can be different at different choice points. Thus, the same literal can have different criticality heuristic scores at different choice points. However, if we calculate the criticality values and the criticality heuristic scores at every choice point, the proportion of the time used by these calculations to total search time will be high. Because of these costly calculations, CSMODELS uses a static heuristic.

In order to limit the amount of backtracking, CSMODELS selects hard-to-choose literals first. So, the system sorts the uncovered atoms according to their heuristic scores. The *sorting criterion* affects the performance of the heuristic. The main sorting criterion of CSMODELS, which is called *maxmin* sorting criterion, is the same as that of SMODELS. Atoms are sorted according to minimum of their positive and negative criticality heuristic scores in descending order. Sorting atoms according to the sum of positive and negative heuristic scores in descending order is another criterion named *maxsum*. Experiments showed that using maxsum helped CSMODELS to find an answer set in time less than using maxmin for problems having many answer sets.

Fortunately, using maxsum in problems that do not have many answer sets usually leads to conflict at the first choice point. CSMODELS first uses the maxsum criterion. If it ends up with an immediate conflict, it switches back to maxmin. This causes CSMODELS to behave identically for all problems.

6 Experimental Results

CSMODELS has been tested to find out its performance. Test problems are taken from common application domains of answer set programming. We compare results of CSMODELS with those of SMODELS. Since both systems take the same ground logic program for the experiments, grounding time is the same for both. The grounding times are not included in the search time results of the experiments.

The main measure for the tests is the duration which states how long the search for an answer set took in CPU seconds. Another measure for comparison is the number of choice points. This number shows how many times a system used its heuristic to find an answer. The same ratio between the number of choice points may not be observed for the search times in comparisons of CSMODELS and SMODELS since there is not a direct correlation. However, the reduction in the number of choice points generally leads to performance gains in terms of search times.

Tests for planning problems have been performed on a 200 MHz Pentium computer with 256 MB of main memory running Linux 2.4.5; for colorability and n-queens problems a 733 MHz Pentium III computer with 256 MB of main memory running Linux 2.2.19 has been used.

Several well-known planning problems are tested. These are Towers of Hanoi, simple robot-box domain, and blocks-world planning. In addition to planning problems, several tests of colorability and n-queens problems have been performed. If the original logic programs for these problems have cardinality rules,

they are rewritten using normal rules without affecting the answer sets. Tables 1, 2 and 3 report the results of CSMODELS and SMODELS for all the problem instances. The results for choice points are shown in parentheses in tables.

Towers of Hanoi problem with 4 disks has been tested. It can be solved in 15 steps optimally. The logic program representing the simple robot–box domain is based on the reference [6].[8] The optimal solution has 13 steps. For the blocks-world planning problem, two different domain representations have been tested. One representation is taken from the reference [17] where concurrency is allowed. Three different problem instances with 15, 17, and 19 blocks from the reference [18] are tested (rows BW_2 (15), BW_2 (17) and BW_2 (19)).[9] They have optimal plans with 8, 9 and 10 steps, respectively. Another representation is the one used in reference [19]. A problem instance with 11 blocks from the same reference has been used (row BW_1 (11)).[10] It has an optimal solution with 9 steps. Unlike the first representation, this one does not allow concurrent moves.

Table 1. A comparison of search time (CPU seconds) and number of choice points for planning problems

	SMODELS	CSMODELS
Hanoi	4.630 (12)	5.540 (8)
Robot–Box	39.940 (11)	40.800 (8)
BW_1 (11)	73.400 (7)	57.040 (5)
BW_2 (15)	38.500 (29)	37.390 (7)
BW_2 (17)	77.330 (27)	61.550 (10)
BW_2 (19)	174.660 (4111)	90.620 (6)

The results show that for small size problems like Towers of Hanoi and simple robot-box domains, CSMODELS' performance is almost the same as SMODELS' performance. When the sizes of the problems become larger, criticality heuristic of CSMODELS helps the system solve the problems more efficiently. This claim is supported especially by the problem instances of the blocks-world representation BW_2. By increasing the number of blocks from 15 to 19, the performance difference between CSMODELS and SMODELS becomes obvious. Also the reduction in the number of choice points is consistent with the performance gains.

Other than the planning domain, colorability (4-colorability problem instances are used in experiments) and n-queens problems have been used in testing. The logic programs for these problems are based on I. Niemela's representations.[11]

[8] This domain is about controlling a robot to move boxes between rooms.

[9] They correspond to I. Niemela's bw-large.c, bw-large.d and bw-large.e problems, respectively [18].

[10] This problem instance corresponds to E. Erdem's P4 [19].

[11] The logic program for n-queens problem is actually from [17]. The one for colorability problem is adapted from [18] by rewriting choice rules.

Table 2. A comparison of search time (CPU seconds) and number of choice points for colorability problem

	p100	*p300*	*p600*	*p1000*	*p3000*
SMODELS	0.530 (32)	3.530 (89)	12.380 (177)	35.210 (310)	305.060 (860)
CSMODELS	0.540 (38)	3.050 (79)	10.080 (162)	26.070 (235)	230.370 (787)

Table 3. A comparison of search time (CPU seconds) and number of choice points for n-queens problem

	8×8	*18×18*	*20×20*	*22×22*
SMODELS	0.020 (3)	1.830 (302)	9.360 (1483)	117.730 (16156)
CSMODELS	0.030 (7)	1.530 (152)	5.490 (571)	31.170 (2401)

Several tests have been carried out ranging from small size problem instances to large sized ones for colorability and n-queens. Results show that CSMODELS is more efficient than SMODELS for colorability and n-queens problems. For larger sized problem instances, the performance gains obtained by CSMODELS are more significant.

7 Conclusion

Answer set programming systems utilize some heuristics to compute answer sets. Heuristics affect the search path that the system explores within the entire search space. As some paths lead to efficient solutions and some not, heuristics are one of the key ingredients influencing the performance of a system.

In our work, a new heuristic for answer set programming has been developed. The resulting system called CSMODELS uses this new heuristic. The main insight of the heuristic comes from hierarchical planning. The notion of criticality, which is introduced for generating abstraction hierarchies, is applied for answer set programming in the new heuristic. The experimental results indicate that CSMODELS outperforms SMODELS in terms of efficiency. The performance difference becomes more significant when the problem size gets larger. Generally, the results show that this new heuristic is promising for answer set programming.

The choice rules from the extended rules of SMODELS' new versions are supported by CSMODELS. However, cardinality and weight rules are not supported. Rewriting cardinality and weight rules using normal rules lead to an exponential growth in the number of rules. Handling cardinality and weight rules in CSMODELS is a topic for future work.

Heuristics for answer set programming are similar to the heuristics developed for satisfiability solvers. Although there is a substantial amount of work done for the latter, there is not much for the former [2]. New heuristics will help systems to be more efficient and make answer set programming more applicable.

References

1. V. Lifschitz. Answer set planning. In *International Conference on Logic Programming*, pages 23–37, 1999.
2. W. Faber, N. Leone, and G. Pfeifer. A comparison of heuristics for answer set programming. In *Proc. of the 5th Dutch-German Workshop on Nonmonotonic Reasoning Techniques and their Applications (DGNMR 2001)*, pages 64–75, 2001.
3. P. Simons. Extending and implementing the stable model semantics. Research Report 58, Helsinki University of Technology, Helsinki, Finland, 2000.
4. A. Bundy, F. Giunchiglia, R. Sebastiani, and T. Walsh. Calculating criticalities. *Artificial Intelligence*, 88(1–2):39–67, 1996.
5. F. Giunchiglia. Using ABSTRIPS abstractions – where do we stand? Technical Report 9607–10, IRST, Trento, Italy, 1996.
6. C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.
7. F. Giunchiglia, A. Villafiorita, and T. Walsh. Theories of abstraction. *AI Communications*, 10(3–4):167–176, 1997.
8. C. A. Knoblock. Abstracting the Tower of Hanoi. In *Working Notes of AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions*, pages 13–23, 1990.
9. D. E. Smith and M. A. Peot. A critical look at Knoblock's hierarchy mechanism. In *Proc. of 1st International conference Artificial Intelligence Planning Systems (AIPS-92)*, pages 307–308, 1992.
10. C. Backstrom and P. Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, pages 1599–1604, 1995.
11. 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. The MIT Press, 1988.
12. T. Syrjanen. LPARSE 1.0 user's manual. Available at http://www.tcs.hut.fi/Software/smodels/lparse.ps.
13. 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.
14. F. Lin and R. Reiter. Rules as actions: A situation calculus semantics for logic programs. *Journal of Logic Programming*, 31(1–3):299–330, 1997.
15. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.
16. O. Sabuncu. Using criticalities as a heuristic for answer set programming. MS Thesis, Middle East Technical University, Department of Computer Engineering, Ankara, Turkey, 2002.
17. I. Niemela and M. Trunszczynski. Answer-set programming: a declarative knowledge representation paradigm. In Lecture notes of ESSLLI 2001 Summer School, 2001.
18. I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.
19. E. Erdem. Applications of logic programming to planning: Computational experiments. Unpublished draft, http://www.cs.utexas.edu/users/esra/papers.html, 1999.

Planning with Preferences Using Logic Programming*

Tran Cao Son and Enrico Pontelli

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
{tson,epontell}@cs.nmsu.edu

Abstract. We present a declarative language, $\mathcal{PP}$, for the specification of preferences between possible solutions (or trajectories) of a planning problem. This novel language allows users to elegantly express non-trivial, multi-dimensional preferences and priorities over them. The semantics of $\mathcal{PP}$ allows the identification of *most preferred trajectories* for a given goal. We also provide an answer set programming implementation of planning problems with $\mathcal{PP}$ preferences.

1 Introduction

Planning—in its classical sense—is the problem of finding a sequence of actions that achieves a predefined goal. Most of the research in AI planning has been focused on methodologies and issues related to the development of efficient planners. To date, several efficient planning systems have been developed (e.g., see [18]). These developments can be attributed to the discovery of good domain-independent heuristics, the use of domain-specific knowledge, and the development of efficient data structures used in the implementation of the planning algorithms. Logic programming has played a significant role in this line of research, providing a declarative framework for the encoding of different forms of knowledge and its effective use during the planning process [24].

However, relatively limited effort has been placed on addressing several important aspects in real-world planning domains, such as *plan quality* and *preferences about plans*. In many real world frameworks, the space of feasible plans to achieve the goal is dense, but many of such plans, even if executable, may present undesirable features. In these frameworks, it may be simple to find a solution; rather, the challenge is to produce a solution that is considered satisfactory w.r.t. the needs and preferences of the user. Thus, feasible plans may have a measure of quality, and only a subset may be considered acceptable. These issues can be illustrated with the following example:

Example 1. It is 7 am and Bob*, a Ph.D. student, is at home. He needs to be at school at 8 am. He can take a bus, a train, or a taxi to go to school, which will take him 55, 45, or 15 minutes respectively. Taking the bus or the train will require* Bob *to walk to the nearby station, which may take 20 minutes. However, a taxi can arrive in only 5 minutes. When in need of a taxi,* Bob *can call either the* MakeIt50 *or the* PayByMeter *taxi company.* MakeIt50 *will charge a flat rate of $50 for any trip, while* PayByMeter *has a fee schedule of $20 for the trip to school. If he takes the bus or the train, then* Bob *will spend only $2.*

* The research has been partially supported by NSF grants EIA0220590, EIA0130887, CCR9875279, CCR9820852, and CCR9900320.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 247–260, 2004.

Bob*, being a student, prefers to pay less whenever possible. It is easy to see that there are only two* feasible *plans for* Bob *to arrive at school on time for his exam: calling one of the two taxi companies. However, a* PayByMeter *taxi would be preferable, as* Bob *wants to save money.* In this case, both plans are feasible but *Bob*'s preference is the deciding factor to select which plan he will follow.

The example shows that users' preferences play a decisive role in the choice of a plan. Thus, we need to be able to evaluate plan components at a finer granularity than simply as consistent or violated. In [20], it is argued that users' preferences are likely more important in selecting a plan for execution, when a planning problem has too many solutions. It is worth observing that, with a few exceptions like the system SIPE-2 with metatheoretic biases [20], most planning systems do not allow users to specify their preferences and use them in finding plans. The responsibility in selecting the most appropriate plan rests solely on the users. It is also important to observe that *preferences* are different from *goals* in a planning problem; a plan *must* satisfy the goal, while it may or may not satisfy the preferences. The distinction is analogous to the separation between *hard* and *soft* constraints [3]. E.g., if Bob's *goal* is to spend at most $2 to go to school, then he does not have any feasible plans to arrive at school on time.

In this paper, we will investigate the problem of integrating users' preferences into a planner. We will develop a high-level language for the specification of user preferences, and then provide a logic programming implementation of the language, based on answer set programming. As demonstrated in this work, normal logic programs with answer set semantics [13] provide a natural and elegant framework to effectively handle planning with preferences. We divide the preferences that a user might have in different categories:

- *Preference about a state:* the user prefers to be in a state s that satisfies a property ϕ rather than a state s' that does not satisfy it, even though both satisfy his/her goal;
- *Preference about an action:* the user prefers to perform the action a, whenever it is feasible and it allows the goal to be achieved;
- *Preference about a trajectory:* the user prefers a trajectory that satisfies a certain property ψ over those that do not satisfy this property;
- *Multi-dimensional Preferences:* the user has a *set* of preferences about the trajectory, with an ordering among them. A trajectory satisfying a more favorable preference is given priority over those that satisfy less favorable preferences.

It is important to observe the difference between ϕ and ψ in the above definitions. ϕ is a *state* property, whereas ψ is a formula over the whole *trajectory* (from the initial state to the state that satisfies the given goal). We will also demonstrate how the language for expressing preferences can be realized using Answer Set Programming (ASP).

Related Work: This work is a continuation of our previous work [25], in which we rely on prioritized default theories to express limited classes of preferences between trajectories. This work is also influenced by other works on exploiting *domain-specific knowledge* in planning (e.g., [2,24]), in which domain-specific knowledge is expressed as a constraint on the trajectories achieving the goal, and hence, is a *hard constraint*.

Numerous approaches have been proposed to integrate preferences in the planning process. Eiter et al. introduced a framework for planning with action costs using logic programming [9]. Each action is assigned an integer cost, and plans with the minimal cost are considered optimal. Costs can be either static or relative to the time step in which the action is executed. [9] also presents the encoding of different preferences, such as shortest

plan and the cheapest plan. Our approach also emphasizes the use of logic programming, but differs in several aspects. Here, we develop a *declarative language* for preference representation. Our language can express the preferences discussed in [9], but it is more high-level and flexible than the action costs approach. The approach in [9] also does not allow the use of fully general dynamic preferences. Other systems have adopted fixed types of preferences, e.g., shortest plans [6,4].

Our proposal has similarities with the approach based on metatheories of the planning domain [19,20], where metatheories provide characterization of semantic differences between the various domain operators and planning variables; metatheories allow the generation of biases to focus the planner towards plans with certain characteristics.

The problem of maintaining and managing preferences has been investigated in the framework of constraint programming (e.g., [3,11]). Constraint solving has also been proposed for the management of planning in presence of action costs [15].

Considerable effort has been invested in introducing preferences in logic programming. In [7] preferences are expressed at the level of atoms and used for parsing disambiguation in logic grammars. Rule-level preferences have been used in various proposals for selection of preferred answer sets in answer set programming [5,8,23].

Our language allows the representation of several types of preferences similar to those developed in [14] for decision-theoretic planners. The main difference is that we use logic programming while their system is probability based. Our approach also differs from the works on using Markov Decision Processes (MDP) to find optimal plans [22]; in MDPs, optimal plans are functions from states to actions, thus preventing the user from selecting preferred trajectories without changing the MDP specification.

2 Preliminary – Answer Set Planning

In this section we review the basics of planning using logic programming with answer set semantics—*Answer Set Planning (or ASP)* [17]. We will assume that the effect of actions on the world and the relationship between fluents in the world are expressed in an appropriate language. In this paper, we will make use of the ontologies of the action description language $\mathcal{B}$ [12]. In $\mathcal{B}$, an action theory is defined over two disjoint sets—the set of actions **A** and the set of fluents **F**; an action theory is a pair (D, I), where D is a set of propositions expressing the effects of actions, the relationship between fluents, and the executability conditions of actions; I is a set of propositions representing the initial state of the world. For example, the action of calling a taxi has the effect of the taxi arriving, and it is represented in $\mathcal{B}$ as: $call\_taxi$ **causes** $taxi\_arrived$. Realistically, should one need to execute this action one has to have enough money. This is expressed in $\mathcal{B}$ by the proposition: $call\_taxi$ **executable_if** has_enough_money. In this paper, we will assume that I is complete, i.e., for every fluent $f \in \mathbf{F}$, I contains either f or $\neg f$.

The semantics of an action theory is given by the notion of a *state*—a consistent set of fluent literals (i.e., fluents and negated fluents) that satisfies the relationship between fluents—and a *transition function* Φ that specifies the result of the execution of an action a in a state s, denoted by $\Phi(a, s)$. A *trajectory* of an action theory (D, I) is a sequence $s_0 a_1 s_1 \ldots a_n s_n$ where s_i's are states, a_i's are actions, and $s_{i+1} \in \Phi(s_i, a_{i+1})$ for $i \in \{0, \ldots, n-1\}$. A state s satisfies a fluent literal f, denoted by $s \models f$, if $f \in s$. Since our main concern in this paper is not the language for representing actions and their effects, we omit here the detailed definition of $\mathcal{B}$ [12].

A planning problem is specified by a triple $\langle D, I, G\rangle$, where (D, I) is an action theory and G is a fluent formula (a propositional formula based on fluent literals) representing the goal. A possible solution to $\langle D, I, G\rangle$ is a trajectory $s_0 a_1 s_1 \ldots a_m s_m$, where $s_0 \models I$ and $s_m \models G$. In this case, we say that the trajectory achieves G.

Answer set planning [17] solves a planning problem $\langle D, I, G\rangle$ by translating it into a logic program $\Pi(D, I, G)$ consisting of *domain-dependent* rules that describe D, I, and G and *domain-independent* rules that generate action occurrences and represent the transitions between states. Besides the planning problem, $\Pi(D, I, G)$ requires an additional parameter: the maximal *length* of the trajectory that the user can accept. The two key predicates of $\Pi(D, I, G)$ are:

- $holds(f, t)$ – the fluent literal f holds at the time moment t; and
- $occ(a, t)$ – the action a occurs at the time moment t.

$holds(f, t)$ can be extended to define $holds(\phi, t)$ for an arbitrary fluent formula ϕ, which states that ϕ holds at the time t. Details about the program $\Pi(D, I, G)$ can be found in [24]. The key property of the translation of $\langle D, I, G\rangle$ into $\Pi(D, I, G)$ is that it ensures that each trajectory achieving G corresponds to an answer set of $\Pi(D, I, G)$, and each answer set of $\Pi(D, I, G)$ corresponds to a trajectory achieving G [24]. Answer sets of the program $\Pi(D, I, G)$ can be computed using answer set solvers such as **smodels** [21], **dlv** [10], **cmodels** [1], or **jsmodels** [16].

3 A Language for Planning Preferences Specification

In this section, we introduce the language $\mathcal{PP}$ for planning preference specification. Let $\langle D, I, G\rangle$ be a planning problem, with actions **A** and fluents **F**; let $\mathcal{F}_F$ be the set of all fluent formulae. $\mathcal{PP}$ is defined as special formulae over **A** and **F**. We subdivide preferences in different classes: *basic desires*, *atomic preferences*, and *general preferences*.

3.1 Basic Desires

A basic desire is a formula expressing a preference about a trajectory. For example, *Bob*'s basic desire is to save money; this implies that he prefers to use the train or the bus to go to school, which, in turn, means that a preferred trajectory for *Bob* should contain the action $take\_bus$ or $take\_train$. This preference could also be expressed by a formula that forbids the fluent $taxi\_arrived$ to become true in every state of the trajectory. These two alternatives of preference representation are not always equivalent. The first one represents the desire of leaving a state by a specific group of actions while the second one represents the desire of being in certain states. Basic desires are constructed by using *state desires* and/or *goal preferences*. Intuitively, a state desire describes a basic user preference to be considered in the context of a specific state. A state desire φ implies that we prefer a state s such that $s \models \varphi$. A state desire $occ(a)$ implies that we prefer to leave state s using the action a. In many cases, it is also desirable to talk about the final state of the trajectory—we call this a *goal preference*. These are formally defined next.

Definition 1 (State Desires and Goal Preferences). *A (primitive) state desire is either a formula φ where $\varphi \in \mathcal{F}_F$, or a formula of the form $occ(a)$ where $a \in$ **A**.*
A goal preference *is a formula of the form **goal**(φ) where φ is a formula in $\mathcal{F}_F$.*

We are now ready to define a basic desire that expresses a user preference over the trajectory. As such, in addition to the propositional connectives $\wedge, \vee, \neg$, we will also use the temporal connectives **next**, **always**, **until**, and **eventually**.

Definition 2 (Basic Desire Formula). *A* Basic Desire Formula *is a formula satisfying one of the following conditions:*

- *a goal preference φ is a basic desire formula;*
- *a state desire φ is a basic desire formula;*
- *given the basic desire formulae φ_1, φ_2, then $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\neg\varphi_1$,* **next**(φ_1), **until**(φ_1, φ_2), **always**(φ_1), *and* **eventually**(φ) *are also basic desire formulae.*

To express that *Bob* would like to take the train or the bus to school, we can write:

$$\textbf{eventually}(occ(take\_bus) \vee occ(take\_train)).$$

If *Bob* does not desire to call a taxi, we can write: **always**$(\neg occ(call\_taxi))$. We could also write: **always**$(\neg taxi\_arrived)$. Note that these encodings have different consequences—the second prevents taxis to be present independently from whether it was called or not.

The definition above is used to develop formulae expressing a desire regarding the structure of trajectories. In the next definition, we will specify when a trajectory satisfies a basic desire formula. In a later section, we will present logic programming rules that can be added to the program $\Pi(D, I, G)$ to compute trajectories that satisfy a basic desire. In the following definitions, given a trajectory $\alpha = s_0 a_1 s_1 \cdots a_n s_n$, the notation $\alpha[i]$ denotes the trajectory $s_i a_{i+1} s_{i+1} \cdots a_n s_n$.

Definition 3 (Basic Desire Satisfaction). *Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots a_n s_n$ be a trajectory, and let φ be a basic desire formula. α satisfies φ (written as $\alpha \models \varphi$) iff*

- $\varphi =$ ***goal***(ψ) *and* $s_n \models \psi$
- $\varphi = \psi \in \mathcal{F}_F$ *and* $s_0 \models \psi$
- $\varphi = occ(a)$, $a_1 = a$, *and* $n \geq 1$
- $\varphi = \psi_1 \wedge \psi_2$, $\alpha \models \psi_1$ *and* $\alpha \models \psi_2$
- $\varphi = \psi_1 \vee \psi_2$, $\alpha \models \psi_1$ *or* $\alpha \models \psi_2$
- $\varphi = \neg\psi$ *and* $\alpha \not\models \psi$
- $\varphi =$ **next**(ψ), $\alpha[1] \models \psi$, *and* $n \geq 1$
- $\varphi =$ **always**(ψ) *and* $\forall(0 \leq i \leq n)$ *we have that* $\alpha[i] \models \psi$
- $\varphi =$ **eventually**(ψ) *and* $\exists(0 \leq i \leq n)$ *such that* $\alpha[i] \models \psi$
- $\varphi =$ **until**(ψ_1, ψ_2) *and* $\exists(0 \leq i \leq n)$ *such that* $\forall(0 \leq j < i)$ *we have that* $\alpha[j] \models \psi_1$ *and* $\alpha[i] \models \psi_2$.

Definition 3 allows us to check whether a trajectory satisfies a basic desire. This will also allow us to compare trajectories. Let us start with the simplest form of trajectory preference, involving a single desire.

Definition 4 (Ordering between Trajectories w.r.t. Single Desire). *Let φ be a basic desire formula and let α and β be two trajectories. The trajectory α is preferred to the trajectory β (denoted as $\alpha \prec_\varphi \beta$) if $\alpha \models \varphi$ and $\beta \not\models \varphi$.*

We say that α and β are indistinguishable (denoted as $\alpha \approx_\varphi \beta$) if one of the two following cases occur: (i) $\alpha \models \varphi$ *and* $\beta \models \varphi$, *or* (ii) $\alpha \not\models \varphi$ *and* $\beta \not\models \varphi$.

Whenever it is clear from the context, we will omit φ from $\prec_\varphi$ and $\approx_\varphi$. We will also allow a weak form of single preference;

Definition 5 (Weak Single Desire Preference). *Let φ be a desire formula and let α, β be two trajectories. α is weakly preferred to β (denoted $\alpha \preceq_\varphi \beta$) iff $\alpha \prec_\varphi \beta$ or $\alpha \approx_\varphi \beta$.*

Proposition 1. *The relation $\preceq_\varphi$ defines a partial order over the trajectories.*

These definitions are also expressive enough to describe a significant portion of preferences that frequently occur in real-world domains. Since some of them are particularly important, we will introduce some syntactic sugars to simplify their use:

- (Strong Desire) given the desire formulae φ_1, φ_2, $\varphi_1 < \varphi_2$ denotes $\varphi_1 \wedge \neg\varphi_2$.
- (Weak Desire) given the desire formulae φ_1, φ_2, $\varphi_1 <^w \varphi_2$ denotes $\varphi_1 \vee \neg\varphi_2$.
- (Enabled Desire) given two actions a_1, a_2, we will denote with $a_1 <^e a_2$ the formula $executable(a_1) \wedge executable(a_2) \Rightarrow occ(a_1) < occ(a_2)$. This can be extended to include disjunction (or group) of actions on each side of the formula.

Definition 6 (Most Preferred Trajectory w.r.t. Single Desire). *Let φ be a basic desire formula. A trajectory α is said to be a* most preferred trajectory *w.r.t. φ, if there is no trajectory β such that $\beta \prec_\varphi \alpha$.*

Note that in the presence of preference, a most preferred trajectory might require extra actions that would have been otherwise considered unnecessary as shown below.

Example 2. Let us enrich the theory from Example 1 with an action called *buy_coffee*, which allows Bob to have coffee. He can do it only at the station. To say that *Bob* prefers to have coffee before he takes the exam, we write: **goal**(*have_coffee*). Any plan satisfying this preference requires *Bob* to stop at the station before taking the exam. E.g., while *calls a taxi* and then *takes the taxi to school* is a valid trajectory for *Bob* to achieve his goal, this is not a most preferred trajectory; instead, *Bob* has to take the taxi to the station, buy the coffee, and then go to school. Besides the action of buy_coffee that is needed for *Bob* to get the coffee, the most preferred trajectory requires the action of *driving to the station*, which is not necessary if *Bob* does not have the preference of having the coffee.

3.2 Atomic Preferences and Chains

Basic desires allow users to specify their preferences and can be used in selecting trajectories which satisfy them. From the definition of a basic desire formula, we can assume that users always have a set of desire formulae and that their desire is to find a trajectory that satisfies all formulae. In many cases, this proves to be too strong and results in situations where no preferred trajectory can be found. For example, *time* and *cost* are often two criteria that a person might have when making a travel plan. This two criteria are often in conflict, i.e., transportation method that takes little time often costs more. It is very unlikely that he/she can get a plan that can satisfy both criteria. Consider Example 1, it is obvious that *Bob* cannot have a plan that costs him only two dollars and allows him to get to destination quickly. To address this problem, we allow a new type of formulae, *atomic preferences*, which represents an ordering between basic desire formulae.

Definition 7 (Atomic Preference). *An* atomic preference formula *is defined as a formula of the type $\varphi_1 \lhd \varphi_2 \lhd \cdots \lhd \varphi_n$ ($n \geq 1$) where $\varphi_1, \ldots, \varphi_n$ are basic desire formulae.*

The intuition behind an atomic preference is to provide an ordering between different desires—i.e., it indicates that trajectories that satisfy φ_1 are preferable to those that satisfy φ_2, etc. Clearly, basic desire formulae are special cases of atomic preferences. The definitions of $\approx$ and $\prec$ are extended to compare trajectories w.r.t. atomic preferences.

Definition 8 (Ordering Between Trajectories w.r.t. Atomic Preferences). *Let α, β be two trajectories, and let $\Psi = \varphi_1 \lhd \varphi_2 \lhd \cdots \lhd \varphi_n$ be an atomic preference formula.*

- *α, β are indistinguishable w.r.t. Ψ (written as $\alpha \approx_\Psi \beta$) if $\forall i.[1 \leq i \leq n \Rightarrow \alpha \approx_{\varphi_i} \beta]$.*
- *α is preferred to β w.r.t. Ψ (written as $\alpha \prec_\Psi \beta$) if $\exists (1 \leq i \leq n)$ such that* **(a)** *$\forall (1 \leq j < i)$ we have that $\alpha \approx_{\varphi_j} \beta$, and* **(b)** *$\alpha \prec_{\varphi_i} \beta$.*

We will say that $\alpha \preceq_\Psi \beta$ if either $\alpha \prec_\Psi \beta$ or $\alpha \approx_\Psi \beta$.

We can show that this version of $\preceq$ is a partial order (with $\approx$ as underlying equivalence).

Proposition 2. *Let Ψ be an atomic preference; then $\preceq_\Psi$ is a partial order.*

A trajectory α is most preferred if there is no other trajectory that is preferred to α.

Example 3. Let us continue with the theory in Example 2. To simplify the representation, we will assume that each action is associated with a degree of safety. We will also write bus, $train$, $taxi_1$, $taxi_2$, and $walk$ to say that Bob takes the bus, train, taxi with *PayByMeter* or *MakeIt50*, or walk, respectively. The following is a desire expressing that *Bob* prefers to get the fastest possible way to go to school:

$$time = \textbf{always}(taxi_1 \vee taxi_2 <^e bus \vee train \vee walk)$$

On the other hand, when he is not in a hurry, *Bob* prefers to get the cheaper way to go to school: $cost = \textbf{always}(walk \vee bus \vee train <^e taxi_1 \vee taxi_2)$

These two preferences can be combined into different atomic preferences, e.g.,

$$time \lhd cost \qquad \text{or} \qquad cost \lhd time.$$

The first one is more appropriate for *Bob* when he is in a hurry while the second one is more appropriate for *Bob* when he has time. The trajectory $\alpha = s_0\, walk\, s_1\, bus\, s_2$ is preferred to the trajectory $\beta = s_0\, call\_taxi(PayByMeter)\, s_1'\, taxi_1\, s_2'$ with respect to the preference $cost \lhd time$, i.e., $\alpha \prec_{cost \lhd time} \beta$. (for brevity, we omit the description of the states s_i's.) However, $\beta \prec_{time \lhd cost} \alpha$.

3.3 General Preferences

A general preference is constructed from atomic preferences as follows.

Definition 9 (General Preferences). *A general preference formula is a formula satisfying one of the following conditions:*

- *An atomic preference Ψ is a general preference;*
- *If Ψ_1, Ψ_2 are general preferences, then $\Psi_1 \& \Psi_2$, $\Psi_1 \mid \Psi_2$, and $!\Psi_1$ are general preferences;*
- *Given a collection of general preferences $\Psi_1, \Psi_2, \ldots, \Psi_k$, then $\Psi_1 \lhd \Psi_2 \lhd \cdots \lhd \Psi_k$ is a general preference.*

Intuitively, the operators $\&, |, !$ are used to express different ways to combine preferences. Syntactically, they are similar to the operations $\wedge, \vee, \neg$ in the construction of basic desire formulae. Semantically, they differ from the operations $\wedge, \vee, \neg$ in a subtle way. For example, given two fluent formulae ϕ and ψ, it is easy to see that both $\phi \vee \psi$ and $\phi \mid \psi$ are general preferences. Although both express our preference over trajectories, the first

formula represents a *single preference* while the second one provides *two different criteria* and we have no preference between them.

Definition 10 (Ordering Between Trajectories w.r.t. General Preferences). *Let Ψ a general preference and α, β two trajectories.*

- *The trajectory α is preferred to β ($\alpha \prec_\Psi \beta$) if:*
  - *Ψ is an atomic preference and $\alpha \prec_\Psi \beta$*
  - *$\Psi = \Psi_1 \& \Psi_2$, $\alpha \prec_{\Psi_1} \beta$ and $\alpha \prec_{\Psi_2} \beta$*
  - *$\Psi = \Psi_1 \mid \Psi_2$ and:* (i) *$\alpha \prec_{\Psi_1} \beta$ and $\alpha \approx_{\Psi_2} \beta$; or* (ii) *$\alpha \prec_{\Psi_2} \beta$ and $\alpha \approx_{\Psi_1} \beta$; or* (iii) *$\alpha \prec_{\Psi_1} \beta$ and $\alpha \prec_{\Psi_2} \beta$*
  - *$\Psi =! \Psi_1$ and $\beta \prec_{\Psi_1} \alpha$ or $\alpha \approx_{\Psi_1} \beta$*
  - *$\Psi = \Psi_1 \lhd \cdots \lhd \Psi_k$, and there exists $1 \leq i \leq k$ such that:* (i) *$\forall (1 \leq j < i)$ we have that $\alpha \approx_{\Psi_j} \beta$, and* (ii) *$\alpha \prec_{\Psi_i} \beta$.*
- *The trajectory α is indistinguishable from the trajectory β ($\alpha \approx_\Psi \beta$) if:*
  - *Ψ is an atomic preference and $\alpha \approx_\Psi \beta$.*
  - *$\Psi = \Psi_1 \& \Psi_2$, $\alpha \approx_{\Psi_1} \beta$, $\alpha \approx_{\Psi_2} \beta$.*
  - *$\Psi = \Psi_1 \mid \Psi_2$, $\alpha \approx_{\Psi_1} \beta$, and $\alpha \approx_{\Psi_2} \beta$.*
  - *$\Psi =! \Psi_1$ and $\alpha \approx_{\Psi_1} \beta$.*
  - *$\Psi = \Psi_1 \lhd \cdots \lhd \Psi_k$, and for all $1 \leq i \leq k$ we have that $\alpha \approx_{\Psi_i} \beta$.*

Again, we can prove that $\approx_\Psi$ is an equivalence relation and $\preceq_\Psi$ is a partial ordering.

Proposition 3. *If Ψ is a general preference, then $\approx_\Psi$ is an equivalence relation.*

Proposition 4. *Let Ψ be a general preference. Then $\prec_\Psi$ is a transitive relation and the relation $\preceq$ is a partial order (with $\approx$ as base equivalence).*

A trajectory α is most preferred if there is no trajectory that is preferred to α.

Example 4. Let us continue with the theory of Example 3. Assume that the safest transportation mode is either the *train* or the expensive *MakeIt50* cab. The preference

$$safety = \textbf{always}(train \vee walk \vee taxi_2 <^e bus \vee taxi_1)$$

says that *Bob* prefers to move around using the safest transportation mode. Further, he prefers safety over time and cost, so we write $safety \lhd (time \& cost)$.

4 Computing Preferred Trajectories

In this section, we address the problem of computing preferred trajectories. The ability to use the operators $\wedge, \neg, \vee$ as well as $\&, |, !$ in construction of preference formulae allows us to combine several preferences into a preference formula. For example, if a user has two atomic preferences Ψ and Φ, but does not prefer Ψ over Φ or vice versa, he can combine them in to a single preference $\Psi \wedge \Phi \lhd \Psi \vee \Phi \lhd \neg\Psi \wedge \neg\Phi$. The same can be done if Ψ or Φ are general preferences. Thus, without lost of generality, we can assume that we only have one preference formula to deal with. Given a planning problem $\langle D, I, G \rangle$ and a preference formula φ, we are interested in finding a preferred trajectory α achieving G for φ. We will show how this can be done in answer set programming. We achieve that by encoding each basic desire φ as a set of rules Π_φ and developing two sets of rules Π_{sat}

and Π_{pref}. Π_{sat} checks whether a basic desire is satisfied by a trajectory. Π_{pref} consist of rules that, when used with the **maximize** construct of **smodels** will allow us to find a most preferred trajectory with respect to a preference formula. Since $\Pi(D, I, G)$ has already been discussed in Section 2, we will begin by defining Π_φ.

4.1 Encoding of Desire Formulae

The encoding of a desire formula is similar to the encoding of a fluent formula in [24]. In our encoding, we will use the predicate $formula$ as a domain predicate. The set $\{formula(l, l) \mid l$ is a fluent literal$\}$ will belong to Π_φ. Each of the atoms in this set declares that each literal is also a formula. Next, each basic desire formula φ will be associated with a unique name n_φ. If φ is a fluent formula then it is encoded by a set of atoms of the form $formula(.,.)$ and is denoted by r_φ. For example, $\varphi = f \wedge g$ will be given a name, $n_{f \wedge g}$, and is encoded by the formula $formula(n_{f \wedge g}, conjunction(f, g))$. For other types of desire formula, Π_φ is defined as follows.

- If $\varphi = goal(\phi)$ then $\Pi_\varphi = r_\phi \cup \{desire(n_\varphi), goal(n_\phi)\}$;
- If φ is a fluent formula then $\Pi_\varphi = r_\varphi \cup \{desire(n_\varphi)\}$;
- If $\varphi = occ(a)$ then $\Pi_\varphi = \{desire(n_\varphi), happen(n_\varphi, a)\}$;
- If $\varphi = \varphi_1 \wedge \varphi_2$ then $\Pi_\varphi = \Pi_{\varphi_1} \cup \Pi_{\varphi_2} \cup \{desire(n_\varphi), and(n_\varphi, n_{\varphi_1}, n_{\varphi_2})\}$;
- If $\varphi = \varphi_1 \vee \varphi_2$ then $\Pi_\varphi = \Pi_{\varphi_1} \cup \Pi_{\varphi_2} \cup \{desire(n_\varphi), or(n_\varphi, n_{\varphi_1}, n_{\varphi_2})\}$;
- If $\varphi = \neg\phi$ then $\Pi_\varphi = \Pi_\phi \cup \{desire(n_\varphi), negation(n_\varphi, n_\phi)\}$;
- If $\varphi =$ **next**(ϕ) then $\Pi_\varphi = \Pi_\phi \cup \{desire(n_\varphi), next(n_\varphi, n_\phi)\}$;
- If $\varphi =$ **until**(φ_1, φ_2) then $\Pi_\varphi = \Pi_{\varphi_1} \cup \Pi_{\varphi_2} \cup \{desire(n_\varphi), until(n_\varphi, n_{\varphi_1}, n_{\varphi_2})\}$;
- If $\varphi =$ **always**(ϕ) then $\Pi_\varphi = \Pi_\phi \cup \{desire(n_\varphi), always(n_\varphi, n_\phi)\}$;
- If $\varphi =$ **eventually**(ϕ) then $\Pi_\varphi = \Pi_\phi \cup \{desire(n_\varphi), eventually(n_\varphi, n_\phi)\}$.

4.2 Π_{sat} – Rules for Checking of Basic Desire Formula Satisfaction

We now present the set of rules that check whether a trajectory satisfies a basic desire formula. Recall that an answer set of the program $\Pi(D, I, G)$ will contain a trajectory where action occurrences are recorded by atoms of the form $occ(a, t)$ and the truth value of fluent literals is represented by atoms of the form $holds(f, t)$, where $a \in \mathbf{A}$, f is a fluent literal, and t is a time moment between 0 and $length$. Π_{sat} defines the predicate $satisfy(F, T)$ where F and T are variables representing a basic desire and a time moment, respectively. The satisfiability of a fluent formula at a time moment will be defined by the predicate $h(F, T)$ – which is defined based on the predicate $holds$ and the usual logical operators such as $\wedge, \vee, \neg$. Intuitively, $satisfy(F, T)$ says that the basic desire F is satisfied by the trajectory starting from the time moment T. They are defined based on the structure of F. Some of the rules of Π_{sat} are given next.

$$satisfy(F, T) \leftarrow desire(F), goal(F), satisfy(F, length). \quad (1)$$

$$satisfy(F, T) \leftarrow desire(F), happen(F, A), occ(A, T). \quad (2)$$

$$satisfy(F, T) \leftarrow desire(F), formula(F, G), h(G, T). \quad (3)$$

$$satisfy(F, T) \leftarrow desire(F), and(F, F_1, F_2), satisfy(F_1, T), satisfy(F_2, T). \quad (4)$$

$$satisfy(F, T) \leftarrow desire(F), negation(F, F_1), not\ satisfy(F_1, T). \quad (5)$$

$$satisfy(F, T) \leftarrow desire(F), until(F, F_1, F_2), during(F_1, T, T_1), satisfy(F_2, T_1). \quad (6)$$

$$satisfy(F,T) \leftarrow desire(F), always(F,F_1), during(F_1,T,length{+}1). \quad (7)$$

$$satisfy(F,T) \leftarrow desire(F), next(F,F_1), satisfy(F_1,T+1). \quad (8)$$

$$during(F_1,T,T_1) \leftarrow T < T_1 - 1, desire(F_1), satisfy(F_1,T), during(F_1,T+1,T_1). \quad (9)$$

$$during(F_1,T,T_1) \leftarrow T = T_1 - 1, desire(F_1), satisfy(F_1,T). \quad (10)$$

In the next theorem, we prove the correctness of Π_{sat}. We need some additional notation. Let M be an answer set of the program $\Pi(D,I,G)$. By α_M we denote the trajectory $s_0 a_0 \ldots a_{n-1} s_n$, where (i) $occ(a_i,i) \in M$ for $i \in \{0,\ldots,n-1\}$ and (ii) $s_i = \{f \mid holds(f,i) \in M\}$ for $i \in \{0,\ldots,n\}$. For a trajectory $\alpha = s_0 a_0 \ldots a_{n-1} s_n$, let $\alpha^{-1} = \{occ(a_i,i) \mid i \in \{0,\ldots,n-1\} \cup \{holds(f,i) \mid f \in s_i, i \in \{0,\ldots,n\}$. We have:

Theorem 1. *Let $\langle D,I,G\rangle$ be a planning problem and φ be a basic desire formula. Let M be an answer set of $\Pi(D,I,G)$. Then, $\alpha_M \models \varphi$ iff $\Pi_\varphi \cup \Pi_{sat} \cup (\alpha_M)^{-1} \models satisfy(n_\varphi, 0)$.*

The theorem allows us to compute a most preferred trajectory in **smodels**. Let $\Pi(D,I,G,\varphi)$ be the program consisting of the $\Pi(D,I,G) \cup \Pi_\varphi \cup \Pi_{sat}$ and the rule

$$\mathbf{maximize}\{satisfy(n_\varphi,0) = 1, not\ satisfy(n_\varphi,0) = 0\}. \quad (11)$$

Note that rule (11) means that answer sets in which $satisfy(n_\varphi,0)$ holds are most preferred. **smodels** will first try to compute answer sets of Π in which $satisfy(n_\varphi,0)$ holds; only when no answer set with this property exists, other answer sets are considered. (The current implementation of **smodels** has some restrictions on using the **maximize** construct; our **jsmodels** system can now deal with this construct properly.)

Theorem 2. *Let $\langle D,I,G\rangle$ be a planning problem and φ be a basic desire formula. For every answer set M of $\Pi(D,I,G,\varphi)$, α_M is a most preferred trajectory w.r.t. φ.*

The above theorem gives us a way to compute a most preferred trajectory with respect to a basic desire. We will now generalize this approach to deal with general preferences using the **maximize** function. The intuition is to associate to the different components of the preference formula a *weight*; these weights are then used to obtain a weight for each trajectory (based on what components of the preference formula are satisfied by the trajectory). The **maximize** function can be used to handle these weights and guide the search of the preferred trajectory. In general, let Ψ be a general preference. We will develop a *weight function*, w_Ψ, which maps each trajectory to a number and satisfies the following properties: **(i)** if $\alpha \prec_\Psi \beta$ then $w_\Psi(\alpha) > w_\Psi(\beta)$, and **(ii)** if $\alpha \approx_\Psi \beta$ then $w_\Psi(\alpha) = w_\Psi(\beta)$. A weight function satisfying the two properties (i)-(ii) is called an *admissible* weight function. Obviously, if w_Ψ is admissible, we have the following theorem.

Proposition 5. *Let Ψ be a general preference formula and $w_\Psi(\alpha)$ be an admissible weight function. If α is a trajectory such that $w_\Psi(\alpha)$ is maximal, then α is a most preferred trajectory w.r.t. Ψ.*

The above proposition implies that we can compute a most preferred trajectory using **smodels** if we can implement an admissible weight function.

4.3 Computing an Admissible Weight Function

Let Ψ be a general preference. We will now show how an admissible weight function w_Ψ can be built in a bottom-up fashion. We begin with basic desires.

Definition 11 (Basic Desire Weight). *Let φ be a basic desire formula and let α be a trajectory. The weight of the trajectory α w.r.t. the desire φ is a function defined as $w_\varphi(\alpha) = 1$ if $\alpha \models \varphi$ and $w_\varphi(\alpha) = 0$, otherwise.*

The next proposition shows that for a basic desire φ, w_φ is admissible.

Proposition 6. *Let φ be a basic desire. Then w_φ is an admissible weight function.*

The weight function of an atomic preference is defined on the weight function of basic desires occurring in the preference as follows.

Definition 12 (Atomic Preference Weight). *Let $\psi = \varphi_1 \lhd \varphi_2 \lhd \cdots \lhd \varphi_k$ be an atomic preference formula. The weight of a trajectory α w.r.t. ψ is defined as follows:*

$$w_\psi(\alpha) = \sum_{r=1}^{k} (2^{k-r} \times w_{\varphi_r}(\alpha))$$

Again, we can show that the above function is admissible.

Proposition 7. *Let $\psi = \varphi_1 \lhd \varphi_2 \lhd \cdots \lhd \varphi_k$ be an atomic preference. Then w_ψ is an admissible weight function.*

We are now ready to define an admissible weight function w.r.t. a general preference.

Definition 13 (General Preference Weight). *Let Ψ be a general preference formula. The weight of a trajectory α w.r.t. Ψ ($w_\Psi(\alpha)$) is defined as follows:*

- *if Ψ is an atomic preference then the weight is defined as in definition 12.*
- *if $\Psi = \Psi_1 \& \Psi_2$ then $w_\Psi(\alpha) = w_{\Psi_1}(\alpha) + w_{\Psi_2}(\alpha)$*
- *if $\Psi = \Psi_1 \mid \Psi_2$ then $w_\Psi(\alpha) = w_{\Psi_1}(\alpha) + w_{\Psi_2}(\alpha)$*
- *if $\Psi = \ !\Psi_1$ then $w_\Psi(\alpha) = max(\Psi_1) - w_{\Psi_1}(\alpha)$ where $max(\Psi_1)$ represents the maximum weight that a trajectory can achieve on the preference formulae Ψ_1.*
- *if $\Psi = \Psi_1 \lhd \Psi_2$[1] then $w_\Psi(\alpha) = max(\Psi_2) \times w_{\Psi_1}(\alpha) + w_{\Psi_2}(\alpha)$*

We prove the admissibility of w_Ψ in the next proposition.

Proposition 8. *If Ψ is a general preference, then w_Ψ is an admissible weight function.*

Propositions 6-8 show that we can compute an admissible weight function w_Ψ bottom-up from the weight of each basic desire occurring in Ψ. We are now ready to define the set of rules $\Pi_{pref}(\Psi)$ which consists of the rules encoding Ψ and the rules encoding the computation of w_Ψ. Similar to the encoding of desires, we will assign a new, distinguished name n_ϕ to each preference formula ϕ, which is not a desire, occurring in Ψ and encode the preferences in the same way we encode the desires. To save space, we omit here the details of this step. $\Pi_{pref}(\Psi)$ define two predicates, $w(p, n)$ and $max(p, n)$ where p is a preference name and n is the weight of the current trajectory with respect to the preference p. $w(p, n)$ (resp. $max(p, n)$) is true if the weight (resp. maximal weight) of the current trajectory with respect to the preference p is n.

1. For each desire d, $\Pi_{pref}(d)$ contains the rules
   $w(d, 1) \leftarrow satisfy(d).$ $\quad w(d, 0) \leftarrow not\ satisfy(d).$ $\quad max(d, 2) \leftarrow .$
2. For each atomic preference $\phi = \varphi_1 \lhd \varphi_2 \lhd \cdots \lhd \varphi_k$, $\Pi_{pref}(\phi)$ consists of $\cup_{j=1}^{k} \Pi_{pref}(\varphi_k)$ and the next two rules:
   $w(n_\phi, S) \leftarrow w(n_{\varphi_1}, N_1), w(n_{\varphi_2}, N_2), \ldots, w(n_{\varphi_k}, N_k), S = \Sigma_1^k 2^{k-r} N_r.$
   $max(n_\phi, 2^{k+1} + 1) \leftarrow .$

[1] Without loss of generality, we describe the encoding only for chains of length 2.

3. For each general preference Ψ,
   - if Ψ is an atomic preference then $\Pi_{pref}(\Psi)$ is defined as in the previous item.
   - if $\Psi = \Psi_1 \& \Psi_2$ or $\Psi = \Psi_1 | \Psi_2$ then $\Pi_{pref}(\Psi)$ consists of $\Pi_{pref}(\Psi_1) \cup \Pi_{pref}(\Psi_2)$ and
     $w(n_\Psi, S) \leftarrow w(n_{\Psi_1}, N_1), w(n_{\Psi_2}, N_2), S = N_1 + N_2.$
     $max(n_\Psi, S) \leftarrow max(n_{\Psi_1}, N_1), max(n_{\Psi_2}, N_2), S = N_1 + N_2.$
   - if $\Psi = !\, \Psi_1$ then $\Pi_{pref}(\Psi)$ consists of $\Pi_{pref}(\Psi_1)$ and the rules
     $w(n_\Psi, S) \leftarrow w(n_{\Psi_1}, N), max(n_{\Psi_1}, M), S = M + 1 - N.$
     $max(n_\Psi, S) \leftarrow max(n_{\Psi_1}, M), S = M + 1.$
   - if $\Psi = \Psi_1 \lhd \Psi_2$ then $\Pi_{pref}(\Psi)$ consists of $\Pi_{pref}(\Psi_1) \cup \Pi_{pref}(\Psi_2)$ and rules
     $w(n_\Psi, S) \leftarrow w(n_{\Psi_1}, N_1), max(n_{\Psi_2}, M_2), w(n_{\Psi_2}, N_2), S = M_2 * N_1 + N_2.$
     $max(n_\Psi, S) \leftarrow max(n_{\Psi_1}, N_1), max(n_{\Psi_2}, N_2), S = N_2 * N_1 + N_2 + 1.$

The next theorem proves the correctness of $\Pi_{pref}(\Psi)$.

Theorem 3. *Let Ψ be a general preference. For every answer set M of $\Pi(D, I, G)$ we have that $\Pi_{pref}(\Psi) \cup \Pi_{sat} \cup (\alpha_M)^{-1} \models w(n_\Psi, w)$ iff $w_\Psi(\alpha_M) = w$.*

The above theorem implies that we can compute a most preferred trajectory by (i) adding $\Pi_{pref}(\Psi) \cup \Pi_{sat}$ to $\Pi(D, I, G)$ and (ii) computing an answer set M in which $w(n_\Psi, w)$ is maximal. A working implementation of this is available in **jsmodels**.

4.4 Some Examples of Preferences in $\mathcal{PP}$

We will now present some preferences that are common to many planning problems and have been discussed in [9]. Let $\langle D, I, G \rangle$ be a planning problem. In keeping with the notation used in the previous section, we use φ to denote G (i.e., $\varphi = G$).

Preference for shortest trajectory: formula based encoding. Assume that we are interested in trajectories achieving φ whose length is less than or equal n. A simple encoding that allows us to accomplish such goal is to make use of basic desires. By $\mathbf{next}^i(\varphi)$ we denote the formula: $\underbrace{\mathbf{next}(\mathbf{next}(\mathbf{next} \cdots (\mathbf{next}}_{i}(\varphi)) \cdots))$.

Let us define the formula $\sigma^i(\varphi)$ $(0 \leq i \leq n)$ as follows:

$$\sigma^0(\varphi) = \varphi \qquad \sigma^i(\varphi) = \bigwedge_{j=0}^{i-1} \neg \mathbf{next}^j(\varphi) \ \wedge \ \mathbf{next}^i(\varphi)$$

Finally, let us consider the formula $short(n, \varphi)$ defined as

$$short(n, \varphi) = \sigma^0(\varphi) \lhd \sigma^1(\varphi) \lhd \sigma^2(\varphi) \lhd \cdots \lhd \sigma^n(\varphi)$$

Proposition 9. *Let α be a most preferred trajectory w.r.t. $short(n, \varphi)$. Then α is a shortest length trajectory satisfying the goal φ.*

Preference for shortest trajectory: action based encoding. The formula based encoding $short(n, \varphi)$ requires the bound n to be given. We now present another encoding that does not require this condition. We introduce two additional fictions actions $stop$ and $noop$ and a new fluent $ended$. The action $stop$ will be triggered when the goal is achieved; $noop$ is used to fill the slot so that we can compare between trajectories; the fluent $ended$ will denote the fact that the goal is achieved. Again, we appeal to the users for the formal representation of these actions. Furthermore, we add the condition $\neg ended$ to the executability condition of any actions in (D, I) and to the initial state I. Then we can encode the condition of shortest length trajectory, denoted by $short$, as

$$\mathbf{always}((stop \vee noop) <^{e} (a_1 \vee \ldots \vee a_k))$$

where $a_1, \ldots, a_k$ are the actions in the original action theory.

Proposition 10. *Let α be a most preferred trajectory w.r.t. short. Then α is a shortest length trajectory satisfying the goal φ.*

Cheapest Plan. Let us assume that we would like to associate a cost $c(a)$ to each action a and determine trajectories that have the minimal cost. Since our comparison is done only on equal length trajectories, we will also introduce the two actions $noop$ and $stop$ with no cost and the fluent $ended$ to record the fact that the goal has been achieved. Further, we introduce the fluent $sCost$ to denote the cost of the trajectory. Initially, we set the value of $sCost$ to 0 and the execution of action a will increase the value of $sCost$ by $c(a)$. The preference $\mathbf{goal}(sCost(m)) \lhd \mathbf{goal}(sCost(m+1)) \ldots \lhd \mathbf{goal}(sCost(M))$ where m and M are the estimated minimal and maximal cost of the trajectories, respectively. Note that we can have $m = 0$ and $M = max\{c(a) \mid a \text{ is an action}\} \times length$.

5 Conclusion

In this paper we presented a novel declarative language, called $\mathcal{PP}$, for the specification of preferences in the context of planning problems. The language nicely integrates with traditional action description languages (e.g., $\mathcal{B}$) and it allows elegant encoding of complex preferences between trajectories. The language provides a *declarative* framework for the encoding of preferences, allowing users to focus on the high-level description of preferences (more than their encoding—as in the approaches based on utility functions). $\mathcal{PP}$ allows the expression of complex preferences, including multi-dimensional preferences. We also demonstrated that $\mathcal{PP}$ preferences can be effectively and easily handled in a logic programming framework based on answer set semantics.

The work is still in its preliminary stages. The implementation of the required cost functions in the **jsmodels** system is almost complete, and this will offer us the opportunity to validate our ideas on large test cases and to compare with related work such as that in [9]. We also intend to explore the possibility of introducing temporal operators at the level of general preferences. These seem to allow for very compact representation of various types of preferences; for example, a shortest plan preference can be encoded simply as:

$$\mathbf{always}((occ(stop) \vee occ(noop)) \lhd (occ(a_1) \vee \ldots \vee occ(a_k)))$$

if $a_1, \ldots, a_k$ are the possible actions.

References

1. Y. Babovich. "CMODELS", `www.cs.utexas.edu/users/tag/cmodels.html`.
2. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1,2):123–191, 2000.
3. S. Bistarelli et al. Labeling and Partial Local Consistency for Soft Constraint Programming. In *Practical Aspects of Declarative Languages*, Springer Verlag, 2000.
4. A.L. Blum and M.L. Furst. Fast Planning through Planning Graph Analysis. *AIJ*, 90, 1997.
5. G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *AIJ*, 109, 1999.
6. A. Cimatti and M. Roveri. Conformant Planning via Symbolic Model Checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.

7. B. Cui and T. Swift. Preference Logic Grammars: Fixed Point Semantics and Application to Data Standardization. *Artificial Intelligence*, 138(1–2):117–147, 2002.
8. J. Delgrande, T. Schaub, and H. Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 3(2):129–187, March 2003.
9. T. Eiter et al. Answer Set Planning under Action Cost. In *JELIA*. Springer Verlag, 2002.
10. T. Eiter et al. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.
11. F. Fages, J. Fowler, and T. Sola. Handling Preferences in Constraint Logic Programming with Relational Optimization. In *PLILP* Springer Verlag, 1994.
12. M. Gelfond and V. Lifschitz. Action languages. *ETAI*, 3(6), 1998.
13. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. ICLP'88.
14. P. Haddawy and S. Hanks. Utility Model for Goal-Directed Decision Theoretic Planners. Technical report, University of Washington, 1993.
15. H. Kautz and J.P. Walser. State-space Planning by Integer Optimization. In *AAAI*, 1999.
16. H. Le, E. Pontelli, T.C. Son. A Java Solver for Answer Set Programming, NMSU, 2003.
17. V. Lifschitz. Answer set planning. In *ICLP*, pages 23–37, 1999.
18. D. Long et al. International Planning Competition.
19. K.L. Myers. Strategic Advice for Hierarchical Planners. In *KR*'96.
20. K.L. Myers and T.J. Lee. Generating Qualitatively Different Plans through Metatheoretic Biases. In *AAAI*, 1999.
21. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *LPNMR*, Springer, pages 420–429, 1997.
22. M.L. Putterman. *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. John Willey & Sons, Inc., New York, NY, 1994.
23. T. Schaub et al. A Comparative Study of Logic Programs with Preferences. *IJCAI*, 2001.
24. T.C. Son, C. Baral, and S. McIlraith. Domain dependent knowledge in planning – an answer set planning approach. In *LPNMR*, Springer, pages 226–239, 2001.
25. T.C. Son and E. Pontelli. Reasoning about actions in prioritized default theory. In *JELIA*, 2002.

Planning with Sensing Actions and Incomplete Information Using Logic Programming*

Tran Cao Son[1], Phan Huy Tu[1], and Chitta Baral[2]

[1] Department of Computer Science
New Mexico State University
PO Box 30001, MSC CS, Las Cruces, NM 88003, USA
{tson,tphan}@cs.nmsu.edu
[2] Department of Computer Science and Engineering Arizona State University
Tempe, AZ 85287, USA
chitta@asu.edu

Abstract. We present a logic programming based conditional planner that is capable of generating both conditional plans and conformant plans in the presence of sensing actions and incomplete information. We prove the correctness of our implementation and show that our planner is complete with respect to the 0-approximation of sensing actions and the class of conditional plans considered in this paper. Finally, we present preliminary experimental results and discuss further enhancements to the program.

1 Introduction

Classical planning assumes that agents have complete information about the world. For this reason, it is often labeled as unrealistic because agents operating in real-world environment often do not have complete information about their environment. Two important questions arise when one wants to remove this assumption: *how to reason about the knowledge of agents* and *what is a plan* in the presence of incomplete information. The first question led to the development of several approaches to reasoning about effects of sensing (or knowledge producing) actions [11,15,16,22,24,26]. The second question led to the notions of *conditional plans* and *conformant plans* which lead to the goal regardless of the value of the unknown fluents in the initial situation. The former contains sensing actions and conditionals such as the well-known "if-then-else" construct and the latter is a sequence of actions. In this paper, we refer to conditional planning and conformant planning as approaches to planning that generate conditional plans and conformant plans, respectively.

Approaches to conditional planning can be characterized by the techniques employed in its search process or by the action formalism that supports its reasoning process. Most of the early conditional planners implement a partial-order planning algorithm [9,10, 20,19,27] and use Situation Calculus or STRIPS as the main vehicle in representing and reasoning about actions and their effects. Among the recent ones, CoPlaS [14] is a

* This work is partially supported by NSF grants EIA-0130887, EIA-0220590, NSF 0070463, and NASA NCC2-1232.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 261–274, 2004.

regression planner implemented in Sicstus Prolog that uses a high-level action description language to represent and reason about effects of actions, including sensing actions; and FLUX [25], implemented in constraint logic programming, is capable of generating and verifying conditional plans. A conditional planner based on a QBF theorem prover was recently developed [21].

Conformant planning [2,3,6,7,23] is another approach to deal with incomplete information. In conformant planning, a solution is a sequence of actions that achieves the goal from every possible initial situation. Recent experimental result shows [3] that conformant planning based on model checking is computationally competitive compared to other approaches to conformant planning such as those based on heuristic search algorithms [2] or those that extends Graphplan [23]. A detailed comparison in [7] demonstrates that a logic programming based conformant planner is competitive with other approaches to planning.

The most important difference between conditional planners and conformant planners lies in the fact that conditional planners can deal with sensing actions and conformant planners cannot. Consequently, there are planning problems solvable by conditional planners but not by conformant planners. Following is one of such examples.

Example 1 (Defusing A Bomb). (From [24]) An agent needs to defuse a bomb. He was given a procedure for completing the job, which works when a special lock on the bomb is in the *off* position (i.e., $locked$). Applying the procedure when the lock is in the $unlocked$ position will cause the bomb to explode and kill the agent. The agent can determine if the lock is locked or not by $looking$ at the lock. He can also $turn$ the lock from the $locked$ position to the $unlocked$ position and vice-versa. He can only execute the above actions if the bomb has not exploded. Initially, the agent knows that the bomb is not defused and is not exploded. His goal is to safely defuse the bomb.

Intuitively, the agent can safely defuse the bomb by **(a)** looking at the lock to determine the position of the lock. He will then know exactly whether the lock is locked or not. If the lock is not locked, he **(b1)** turns the lock to the locked position and applies the procedure to defuse the the bomb; otherwise, he simply needs to **(b2)** apply the procedure to defuse the bomb.

Observe that no sequence of actions can achieve the goal from every possible initial situation, i.e., *there exists no conformant plan* that can help the agent to achieve his goal.

In this paper, we investigate the use of *answer set programming* in conditional planning. Given a planning problem with sensing actions and incomplete information about the initial situation, we translate it into a logic program whose answer sets – which can be computed using the well-known answer set solvers **smodels** [17] or **dlv** [5] – correspond to conditional plans satisfying the goal. We discuss the advantages and disadvantages of the approach and provide the results of our initial experiments. We will review the basics of an action language with sensing actions in the next section. Afterward, we present the main result of this paper, a logic programming encoding of a conditional planner with sensing actions. We then discuss the properties of our logic program. Finally, we discuss some desirable extensions of the current work.

2 Preliminaries

In this paper, we use a variation of the language $\mathcal{A}_K$ with its 0-approximation [24] as the representation language for our planner. We adopt the 0-approximation of $\mathcal{A}_K$ as the complexity of the planning problem with respect to this approximation (under limited sensing) is **NP**-complete [1] which is lower than the complexity of the problem with respect to the $\mathcal{A}_K$ semantics. This allows us to use different systems capable of computing answer sets of logic programs in our implementation. $\mathcal{A}_K$ extends the high-level action description language $\mathcal{A}$ of Gelfond and Lifschitz [8] by introducing two new types of propositions, the *knowledge producing proposition* and the *executability condition*. In this paper, we will also allow *static causal laws* and sensing actions which determine the truth value of more than one fluent with the restriction that the action theories are deterministic (precise definition follows shortly). We now review the basics of the language $\mathcal{A}_K$ and discuss the notion of a conditional plan.

2.1 Action Language $\mathcal{A}_K$ – Syntax

Propositions in $\mathcal{A}_K$ are of the following form:

$$\textbf{causes}(a, f, \{p_1, \ldots, p_n\}) \tag{1}$$
$$\textbf{executable}(a, \{p_1, \ldots, p_n\}) \tag{2}$$
$$\textbf{if}(f, \{p_1, \ldots, p_n\}) \tag{3}$$
$$\textbf{determines}(a, \{l_1, \ldots, l_m\}) \tag{4}$$
$$\textbf{initially}(f) \tag{5}$$

where f and p_i's are fluent literals (a *fluent literal* is either a fluent g or its negation $\neg g$), each l_i is a fluent literal, and a is an action.

(1) represents the conditional effect of action a while (2) states an executability condition of a. A *static causal law* is of the form (3) and states that whenever $p_1, \ldots, p_n$ hold then f must hold. A proposition of the form (4) states that the value of fluent literals l_i's, called *sensed-fluent literals*, will be known after a is executed. We will require that the l_i's are mutually exclusive, i.e., in every state of the real world at most one of the l_i's holds. Propositions of the form (5), also called *v-propositions*, are used to describe the initial situation. An action theory is given by a pair (D, I) where D consists of propositions of the form (1)-(4) and I consists of propositions of the form (5). D and I will be called the *domain description* and *initial situation*, respectively.

Actions occurring in (1) and (4) are called non-sensing actions and sensing actions, respectively. In this paper, we assume that the set of sensing actions and the set of non-sensing actions are disjoint. We will also assume that each sensing action occurs in only one proposition of the form (4).

Example 2. The domain in Example 1 can be represented by the following action theory:

$$D_1 = \left\{ \begin{array}{ll} \textbf{causes}(disarm, exploded, \{\neg locked\}) & \textbf{if}(dead, \{exploded\}) \\ \textbf{causes}(disarm, disarmed, \{locked\}) & \textbf{executable}(disarm, \{\neg exploded, \neg dead\}) \\ \textbf{causes}(turn, \neg locked, \{locked\}) & \textbf{executable}(turn, \{\neg exploded, \neg dead\}) \\ \textbf{causes}(turn, locked, \{\neg locked\}) & \\ \textbf{determines}(look, \{locked, \neg locked\}) & \textbf{executable}(look, \{\neg exploded, \neg dead\}) \end{array} \right\}$$

$$I_1 = \{ \textbf{initially}(\neg dead) \ \ \textbf{initially}(\neg disarmed) \ \ \textbf{initially}(\neg exploded) \}$$

2.2 Conditional Plan

In the presence of incomplete information and sensing actions, we need to extend the notion of a plan from a sequence of actions so as to allow conditional statements such as if-then-else, while-do, or case-endcase (see e.g. [12,15,24]). Notice that an if-then-else statement can be replaced by a case-endcase statement. Moreover, if we are interested only in plans with bounded length, then whatever can be represented by a while-do statement with a non-empty body could be represented by a set of case-endcase statements. Therefore, in this paper, we limit ourselves to conditional plans with only the case-endcase construct. Formally, we consider conditional plans defined as follows.

Definition 1 (Conditional Plan).

1. *A sequence of non-sensing actions $a_1; \ldots; a_k$ ($k \geq 0$), is a conditional plan.*
2. *If $a_1; \ldots; a_{k-1}$ is a non-sensing action sequence, a_k is a sensing action that determines $f_1, \ldots, f_n$, and $c_1, \ldots, c_n$ are conditional plans, then $a_1; \ldots; a_{k-1}; a_k; case(\{f_i \rightarrow c_i\}_{i=1}^n)$ is a conditional plan.*
3. *Nothing else is a conditional plan.*

To execute a plan $a_1; \ldots; a_{k-1}; a_k; case(\{f_i \rightarrow c_i\}_{i=1}^n)$, the agent first executes $a_1, \ldots a_k$. It then evaluates f_i with respect to its knowledge. If it knows that one of the f_i is true it executes c_i. If none of the f_i's is true, then the plan fails and the execution of the conditional plan which contains this conditional plan also fails. It is easy to see that the following conditional plan achieves the goal of the agent in Example 1:

$$c_1 = look;\ case(\{locked \rightarrow disarm,\ \neg locked \rightarrow turn; disarm\}).$$

2.3 0-Approximation of Semantics of $\mathcal{A}_K$

The 0-approximation of an action theory (D, I) is defined by a transition function Φ that maps pairs of states and actions into sets of states. An *a-state* is a pair $\langle T, F\rangle$, where T and F are disjoint sets of fluents. Intuitively, T (resp. F) is the set of fluents which are known to be true (resp. known to be false) in $\langle T, F\rangle$.

Given a fluent f and an a-state $\sigma = \langle T, F\rangle$, we say that a fluent f is *true* (resp. *false*) in σ if $f \in T$ (resp. $f \in F$); and f is *known* (resp. *unknown*) in σ if $f \in T \cup F$ (resp. $f \notin T \cup F$). A positive (resp. negative) fluent literal f is said to *hold* in σ if $f \in T$ (resp. $\bar{f} \in F$). A set of fluent literals X holds in σ if each $l \in X$ holds in σ. An a-state is *complete* if for every fluent f of the theory, $f \in T \cup F$; it is incomplete

otherwise. An a-state σ satisfies **if**$(f, \{f_1, \ldots, f_n\})$ if it is the case that if $\{f_1, \ldots, f_n\}$ holds in σ then f holds in σ. The truth value of fluent formula in σ is defined in the standard way, e.g., $f \wedge g$ (resp. $f \vee g$) holds in σ iff f holds in σ and (resp. or) g holds in σ. A *state* in (D, I) is an a-state that satisfies all static causal laws of the theory.

A state $\sigma = \langle T, F\rangle$ is said to extend an a-state $\sigma' = \langle T', F'\rangle$, denoted by $\sigma' \preceq \sigma$, iff $T' \subseteq T$ and $F' \subseteq F$; in addition, if $T' \neq T$ or $F' \neq F$, we say that σ properly extends σ' and write $\sigma' \prec \sigma$. If $\sigma' \preceq \sigma$, and there exists no state σ^+ such that $\sigma' \preceq \sigma^+ \prec \sigma$, we say that σ *minimally extends* σ'. For an a-state σ, by $Cl_D(\sigma)$ we denote the set of states of D that minimally extends σ. Notice that $Cl_D(\sigma)$ might be empty. It is also possible that $Cl_D(\sigma)$ contains more than one elements. In this case, we say that D is *non-deterministic*. Since our main goal in this paper is to generate plans in the presence of sensing actions and incomplete knowledge about the initial situation, we will assume that action theories in consideration are deterministic, i.e., given an a-state σ, $|Cl_D(\sigma)| \leq 1$.

Given an action a and a state σ, $\Phi(a, \sigma)$ denotes the set of states that may be reached after the execution of a in σ. An action a is executable in a state σ if D contains a proposition **executable**$(a, \{p_1, \ldots, p_n\})$ and p_i's hold in σ. Executing a sensing-action a in a state will cause one of the sensed-fluent literals that occur in the proposition of the form (4) containing a to be true. Executing a non-sensing action a in a state σ will cause some fluents to become true, some become false, *some may become true*, and *some may become false*[1]. These four sets of fluents are denoted by $e_a^+(\sigma)$, $e_a^-(\sigma)$, $F_a^+(\sigma)$, and $F_a^-(\sigma)$, respectively. The result function Res of D is defined by $Res(a, \langle T, F\rangle) = \langle (T \cup e_a^+) \setminus F_a^-, (F \cup e_a^-) \setminus F_a^+\rangle$.

Definition 2 (Transition Function). *Given a domain description D, and action a and a state $\sigma = \langle T, F\rangle$, $\Phi(a, \sigma)$ is defined as follows:*

- *If a is not executable in σ then $\Phi(a, \sigma) = \{\bot\}$, where $\bot$ denotes an undefined state.*
- *If a is executable in σ and a is a non-sensing action then*

$$\Phi(a, \sigma) = \begin{cases} \{\bot\} & \text{if } Cl_D(Res(a, \sigma)) = \emptyset \\ Cl_D(Res(a, \sigma)) & \text{otherwise} \end{cases}$$

- *If a is executable in σ and a is a sensing action that occurs in a proposition of the form* **determines**$(a, \{l_1, \ldots, l_k\})$ *then*

$$\Phi(a, \sigma) = \begin{cases} \{\bot\} & \text{if } Cl_D(\langle T, F\rangle \cup \{l_i\}) = \emptyset \text{ for some } i \\ \bigcup_{i=1}^{k} Cl_D(\langle T, F\rangle \cup \{l_i\}) & \text{if } Cl_D(\langle T, F\rangle \cup \{l_i\}) \neq \emptyset \text{ for all } i's \\ & \text{and none of the } l_j's \text{ is known in } \sigma \\ \{\sigma\} & \text{otherwise} \end{cases}$$

where, for a fluent l, $\langle T, F\rangle \cup \{l\} = \langle T \cup \{l\}, F\rangle$ and $\langle T, F\rangle \cup \{\neg l\} = \langle T, F \cup \{l\}\rangle$.

We say that a domain description D is *consistent* if for every pair of a state σ and an action a, if a is executable in σ then $\bot \notin \Phi(a, \sigma)$. The extended transition function $\hat{\Phi}$ which maps pairs of conditional plans and states into sets of states is defined next.

[1] Space limitation does not allow us to include a detailed review of the 0-approximation of $\mathcal{A}_K$ (see [24]).

Definition 3 (Extended Transition Function). *Given a state σ and a conditional plan c, $\hat{\Phi}(c,\sigma)$ is defined as follows:*

- *If $c = []$ then $\hat{\Phi}(c,\sigma) = \{\sigma\}$,*
- *If $c = a_1;\ldots;a_k$, where $k > 0$ and a_i's are non-sensing actions then $\hat{\Phi}(c,\sigma) = \bigcup_{\sigma' \in \Phi(a_1,\sigma)} \hat{\Phi}([a_2;\ldots;a_k],\sigma')$,*
- *If $c = a; case(\{f_i \rightarrow c_i\}_{i=1}^n)$ and a is a sensing action then $\hat{\Phi}(c,\sigma) = \bigcup_{\sigma' \in \Phi(a,\sigma)} \hat{E}(case(\{f_i \rightarrow c_i\}_{i=1}^n),\sigma')$ where*

$$\hat{E}(case(\{f_i \rightarrow c_i\}_{i=1}^n),\gamma) = \begin{cases} \hat{\Phi}(c_i,\gamma) & \text{if } f_i \text{ holds in } \gamma \\ \{\bot\} & \text{if none of the } f_i \text{ holds in } \gamma, \end{cases}$$

- *If $c = a_1;\ldots,a_k; case(\{f_i \rightarrow c_i\}_{i=1}^n)$ and $k > 1$ then $\hat{\Phi}(c,\sigma) = \bigcup_{\sigma' \in \hat{\Phi}([a_1;\ldots,a_{k-1}],\sigma)} \hat{\Phi}([a_k; case(\{f_i \rightarrow c_i\}_{i=1}^n)],\sigma')$,*
- *For every conditional plan c, $\hat{\Phi}(c,\bot) = \{\bot\}$.*

For an action theory (D,I), let $T_0 = \{f \mid f \text{ is a fluent, } \textbf{initially}(f) \in I\}$ and $F_0 = \{f \mid f \text{ is a fluent, } \textbf{initially}(\neg f) \in I\}$. A state σ is called an *initial state* of (D,I) if $\sigma \in Cl_D(\langle T_0, F_0\rangle)$. A model is a pair (σ_0, Φ) where σ_0 is an initial state and Φ is the transition function of (D,I). The entailment relation of (D,I) is defined next.

Definition 4. *Let (D,I) be a consistent action theory, c be a conditional plan, and ϕ be a fluent formula. We say $D \models_I \phi$* **after** *c if for every model (σ_0,Φ) of (D,I) it holds that $\bot \notin \hat{\Phi}(c,\sigma_0)$ and ϕ holds in every state σ belonging to $\hat{\Phi}(c,\sigma_0)$.*

Example 3. For the action theory (D_1, I_1) in Example 2, we have that

$$D_1 \models_{I_1} \neg dead \wedge \neg exploded \wedge disarmed \textbf{ after } c_1$$

where $c_1 = look;\ case(\{locked \rightarrow disarm,\ \neg locked \rightarrow turn; disarm\})$.

3 A Logic Programming Based Conditional Planner

We now present a logic programming based conditional planner. Given a planning problem $\mathcal{P} = (D,I,G)$, where (D,I) is an action theory and G is a conjunction of fluent literals, representing the goal, we will translate $\mathcal{P}$ into a logic program $\pi(\mathcal{P})$ whose answer sets represent solutions to $\mathcal{P}$. Our intuition behind this task rests on an observation that each conditional plan c (Definition 1) corresponds to a labeled plan tree T_c (Figure 1) defined as follows.

- For $c = a_1;\ldots;a_k$, where a_i's are non-sensing actions, T_c is a tree with k nodes labeled $a_1,\ldots,a_k$, respectively, and a_{i+1} is the child of a_i for $i = 1,\ldots,k-1$. If $c = []$, T_c is the tree with only one node, whose label is nil (the empty action). The label for each link of T_c is empty string.
- For $c = a_1;\ldots;a_k; case(\{f_1 \rightarrow c_i\}_{i=1}^n)$, where a_k is a sensing action that determines f_i's and other a_i's are non-sensing actions, T_c is a tree whose root has the label a_1, a_i has a child with the label a_{i+1} for $i = 1,\ldots,k-1$, a_k has n children, the i^{th} child (from left to right) is the root of T_{c_i} and the label of the link is f_i. The label of the link between a_i and a_{i+1} $(1 \leq i \leq k-1)$ is empty.

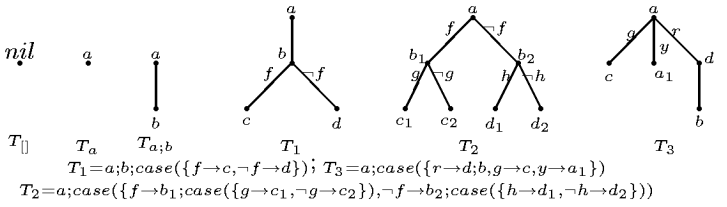


Fig. 1. Sample plan trees

For a conditional plan c, let w be the number of leaves of T_c and h be the number of nodes along a longest path between the root and nodes of T_c. w and h will be called the *width* and *height* of T_c respectively. Let us denote the leaves of T_c by $l_1, \ldots, l_w$. We map each node k of T_c to a pair of integers n_k = (t_k,p_k), where $1 \leq t_k \leq h$ and $1 \leq p_k \leq w$, in the following way:

- For the i-th leaf l_i of T_c, $n_{l_i} = (t_{l_i}, i)$ where t_{l_i} is the number of nodes along the path from l_i to the root of T_c.
- For each interior node k of T_c with children $k_1, \ldots, k_r$ where $n_{k_j} = (t_{k_j}, p_{k_j})$, $n_k = (\min\{t_{k_1}, \ldots, t_{k_r}\}, \min\{p_{k_1}, \ldots, p_{k_r}\})$.

From the construction of T_c, it is easy to see that, independent of how we number the leaves of T_c, we have that

1. If $k_1, \ldots, k_r$ are the children of k with $n_{k_j} = (t_{k_j}, p_{k_j})$, then $t_{k_i} = t_{k_j}$ for every pair of i, j, $1 \leq i, j \leq r$.
2. The root of T_c is always mapped to the pair $(1, 1)$.

One of the possible mappings for the trees in Figure 1 is given in Figure 2.

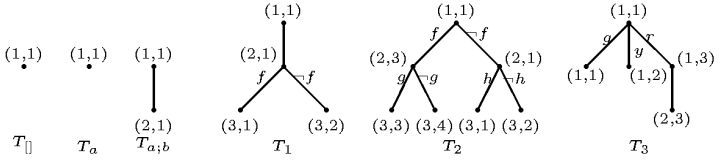


Fig. 2. A possible mapping for the trees in Figure 1

The above observation suggests us to add one more parameter to encode conditional plans. In our representation, besides the maximal length of the plan (the height of the tree), denoted by $length$, we have another parameter denoting its maximal width, denoted by $level$. A plan is encoded on the grid $length \times level$. Action occurrences will happen at the nodes of the grid and links connecting different paths representing case statements. The root of the tree will be at the node $(1, 1)$. We use variables of the type $time$ and $path$, respectively, to denote the position of a node relatively to the root. Instead of the

predicate $holds(F, T)$ (see e.g. [4,13]) that the fluent literal F holds at the time T, we use $h(F, T, P)$ ($unknown(F, T, P)$) to represent the fact that F is true (unknown) at the node (T, P) (the time moment T, on the path number P).

We now describe the program $\pi(\mathcal{P})$. Due to the fact that we use the **smodels** system in our experiment, $\pi(\mathcal{P})$ contains some rules that are specific to **smodels** such as the choice rule. In describing the rules of $\pi(\mathcal{P})$, the following convention is used: T denotes a variable of the sort $time$, $P, P_1, \ldots$ denote variables of the sort $path$, F, G are variables denoting *fluent literals*, $S, S_1, \ldots$ denote variables representing sensed-fluent literals, and A, B are variables representing $action$. We will also write $\bar{F}$ to denote the contrary of F. The main predicates of $\pi(\mathcal{P})$ are listed below:

- $h(L, T, P)$ is true if literal L holds at (T, P),
- $possible(A, T, P)$ is true if action A is executable at (T, P),
- $occ(A, T, P)$ is true if action A occurs at (T, P),
- $used(T, P)$ is true if starting from T, the path P is used in constructing the plan,
- $ph(L, T, P)$ is true if literal L possibly holds at (T, P), and
- $br(S, T, P, P_1)$ is true if node $(T + 1, P_1)$ is a child of (T, P) where S (a sensed-fluent literal) holds at $(T + 1, P_1)$.

The main rules, divided into different groups, are given next. Notice that they are just shown in an abstract form in which atoms indicating the sort of variables, such as $time, path, literal$, and $sense$, that occur in the rule are omitted. For example, if a rule contains a variable T of sort $time$, it should be understood implicitly that the body of the rule also contains the atom $time(T)$. Similarly, if $\bar{F}$, where F is a literal, appears in the rule, then its body also contains the two atoms $literal(F)$ and $contrary(F, \bar{F})$

- *Rules encoding the initial situation.* If **initially**$(f) \in I$, $\pi(\mathcal{P})$ contains the fact

$$h(f, 1, 1) \leftarrow \tag{6}$$

  This says that if **initially**$(f) \in I$ then f must hold at $(1, 1)$, the root of our tree[2].
- *Rule for reasoning about action's executability.* For each proposition of the form (2), $\pi(\mathcal{P})$ contains the rule

$$possible(a, T, P) \leftarrow h(p_1, T, P), \ldots, h(p_n, T, P). \tag{7}$$

- *Rule for reasoning with static causal laws.* For each proposition of the form (3), $\pi(\mathcal{P})$ contains the rule

$$h(f, T, P) \leftarrow h(p_1, T, P), \ldots, h(p_n, T, P). \tag{8}$$

- *Rules for reasoning about the effects of non-sensing actions.* For each proposition of the form (1), we add to $\pi(\mathcal{P})$ the following rules:

$$h(f, T{+}1, P) \leftarrow occ(a, T, P), possible(a, T, P), h(p_1, T, P), \ldots, h(p_n, T, P). \tag{9}$$

$$ab(\overline{f}, T, P) \leftarrow occ(a, T, P), possible(a, T, P), ph(p_1, T, P), \ldots, ph(p_n, T, P). \tag{10}$$

$$h(F, T{+}1, P) \leftarrow h(F, T, P), not\ h(\overline{F}, T{+}1, P), not\ ab(F, T, P). \tag{11}$$

$$ph(F, T, P) \leftarrow not\ h(\overline{F}, T, P). \tag{12}$$

[2] Recall that the root of the tree always has the label $(1, 1)$.

Here, a is a non-sensing action. It changes the world according to the Res function. The first rule encodes the effects of a containing in $e_a^+(\sigma)$ and $e_a^-(\sigma)$ while the second rule indicates that the value of f could potentially be changed by the action a. This rule, when used in conjunction with the inertial rule (11), will have the effect of removing what will possibly become false $F_a^-(\sigma)$ (or true $F_a^+(\sigma)$) from σ, the current state.

– *Rules for reasoning about the effects of sensing actions.* For each proposition **determines**$(a, \{l_1, \ldots, l_n\})$ in D, $\pi(\mathcal{P})$ contains the set of facts $\{sense(l_i) \mid i = 1, \ldots, n\}$ and the following rules:

$$br(l_1, T, P, P) \leftarrow occ(a, T, P), possible(a, T, P). \quad (13)$$

$$1\{br(l_i, T, P, P_1) : new\_br(P, P_1)\}1 \leftarrow occ(a, T, P), possible(a, T, P). \quad (14)$$

$$\text{(for } i = 2, \ldots, n)$$

$$\leftarrow occ(a, T, P), known(l_i, T, P). \quad (15)$$

The first two rules, in conjunction with the rules (20)–(22), state that when a sensing action a that senses the fluent literals l_i's occurs at the node (T, P), $n - 1$ new branches will be created and on each of the branch, one and only one of the sensed-fluent literal will become true. The last rule is a constraint that prevents a from occurring if one of the l_i's is already known. We note that the rule (14) has a different syntax than normal logic programming rule. They are introduced in [18] to ease the programming with **smodels**. The next rules describe the effect of the execution of a sensing actions.

$$new\_br(P, P_1) \leftarrow P < P_1. \quad (16)$$

$$used(T + 1, P_1) \leftarrow br(S, T, P, P_1). \quad (17)$$

$$h(S, T{+}1, P_1) \leftarrow br(S, T, P, P_1). \quad (18)$$

$$h(G, T{+}1, P_1) \leftarrow br(S, T, P, P_1), h(G, T, P). \quad (19)$$

$$\leftarrow S \neq S_1, br(S, T, P_1, P_2), br(S_1, T, P_3, P_2). \quad (20)$$

$$\leftarrow P_3 \neq P_1, br(S_1, T, P_1, P2), br(S_2, T, P_3, P2). \quad (21)$$

$$\leftarrow T > 0, P_1 \neq P_2, br(S, T, P_1, P_2), used(T, P_2). \quad (22)$$

Rule (16) and the last three constraints make sure that none of the already in-use branches is selected when a new branch is created. The next two rules record that the path P is used starting from $(T + 1, P)$ and that S will be true at $(T + 1, P)$ when the literal $br(S, T, P_1, P)$ is true. Rule (19) plays the role of the inertial rule with respect to sensing actions. It makes sure whatever holds at (T, P) will continue to hold at $(T{+}1, P)$.

– *Rules for reasoning about what is known/unknown.*

$$unknown(F, T, P) \leftarrow not\ known(F, T, P). \quad (23)$$

$$known(F, T, P) \leftarrow h(F, T, P). \quad (24)$$

$$known(F, T, P) \leftarrow h(\overline{F}, T, P). \quad (25)$$

Rules of this group are rather standard. They say that if a fluent is true (or false) at (T, P) then it is known at (T, P). Otherwise, it is unknown.

– *Rules for generating action occurrences.*

$$1\{occ(A,T,P) : action(A)\}1 \leftarrow used(T,P), not\ sgoal(T,P), \quad (26)$$
$$\leftarrow occ(A,T,P), not\ possible(A,T,P). \quad (27)$$

The first rule is a choice rule that makes sure that only one action is executed at a node of the tree where the goal has not been achieved. The next rule requires that an action is executed only when it is executable.

– *Auxiliary Rules.*

$$literal(G) \leftarrow fluent(G). \quad (28)$$
$$literal(\neg G) \leftarrow fluent(G). \quad (29)$$
$$contrary(F, \neg F) \leftarrow fluent(F). \quad (30)$$
$$contrary(\neg F, F) \leftarrow fluent(F). \quad (31)$$
$$used(1,1) \leftarrow \quad (32)$$
$$used(T{+}1,P) \leftarrow used(T,P). \quad (33)$$
$$sub\_goal(T,P) \leftarrow not\ not\_sub\_goal(T,P). \quad (34)$$
$$not\_sub\_goal(T,P) \leftarrow finally(F), not\ h(F,T,P). \quad (35)$$
$$\leftarrow used(length,P), not\ sub\_goal(length,P). \quad (36)$$

The first two rules define what is a fluent literal. The next two rules specify that F and $\neg F$ are contrary literals. Rules (32) and (33) mark what nodes are used. Finally, rules (35) and (36) represent the fact that the goal must be achieved at every path of the plan-tree.

4 Properties of $\pi(\mathcal{P})$

In previous answer set based planners [4,6,7,13], reconstructing a plan from an answer set of the program encoding the planning problem is simple: we only need to collect the action occurrences belonging to the model, order them by the time they occur, and we have a plan, i.e., if the answer set contains $occ(a_1, 1)$, ..., $occ(a_n, n)$ then the plan is $a_1, \ldots, a_n$. For $\pi(\mathcal{P})$, the reconstruction process is not that simple because each answer set of $\pi(\mathcal{P})$ represents a conditional plan which may contain case-statements. These conditionals, as we have discussed before, are represented by atoms of the form $br(F,T,P,P_1)$. Thus we have one more dimension (the path number) to deal with and we also need to consider the occurrences of branching literals of the form $br(F,T,P,P_1)$. Let $\mathcal{P} = (D,I,G)$ be a planning problem and S be an answer set of $\pi(\mathcal{P})$, and i,k be integers. We define:

$$p_i^k(S) = a_{i,k}; \ldots; a_{i+l-1,k}; a_{i+l,k}; \textit{case}(\{l_j \rightarrow p_{i+l+1}^{k_j}(S)\}_{j=1}^{t})$$

where $0 \leq l$, $a_{i,k}, \ldots, a_{i+l-1,k}$ are non-sensing actions, $a_{i+l,k}$ is a sensing action with the proposition **determines**$(a_{i+1,k}, \{l_1, \ldots, l_t\})$ in D, S contains the action occurrences $occ(a_{j,k}, t, k)$ for $j = i, \ldots, i+l$ and the branching literals $br(l_j, i+l, k, k_j)$ for $j = 1, \ldots, t$; $p_i^k(S) = []$ if S does not contain some atoms of the form $occ(a, i_1, k)$ for $i_1 \geq i$. Intuitively, $p_i^k(S)$ is the conditional plan with the root at (i,k).

We will subsequently prove that $p_1^1(S)$ is a solution to the planning problem $\mathcal{P}$. First, we prove that $\pi(\mathcal{P})$ correctly implements the transition function Φ (Lemma 1) and no branching is made when non-sensing actions occur whereas branching is required when sensing actions occur. Assume that S is an answer set of $\pi(\mathcal{P})$, we define $s_{i,k}{=}\langle T, F\rangle$ where $T{=}\{f \mid f$ is a fluent, $h(f,i,k) \in S\}$ and $F{=}\{f \mid f$ is a fluent, $h(\neg f,i,k) \in S\}$.

Lemma 1. *Let S be an answer set of $\pi(\mathcal{P})$ whose input parameters are $length$ and $level$, i,k be integers, and $occ(a,i,k) \in S$. Then,*

1. *if a is a non-sensing action then a is executable in $s_{i,k}$, $s_{i+1,k} \in \Phi(a,s_{i,k})$, and $br(f,i,k,k_1) \notin S$ for every pair of a fluent f and an integer k_1; and*
2. *if a is a sensing action with the proposition* **determines**$(a,\{l_1,\ldots,l_m\})$ *in D, then a is executable in $s_{i,k}$ and for every j, $1 \leq j \leq m$,*
   - *$unknown(l_j,i,k) \in S$, and*
   - *if $j > 1$, there exists some integer $k_j \leq level$ such that $used(i,k_j) \notin S$, and $br(l_j,i,k,k_j) \in S$, and $\Phi(a,s_{i,k}) = \{s_{i+1,k}\} \cup \{s_{i+1,k_2},\ldots,s_{i+1,k_m}\}$.*

With the help of the above lemma we can prove the following theorem.

Theorem 1. *Let $\mathcal{P} = (D,I,G)$ be a planning problem and S be an answer set of $\pi(\mathcal{P})$. Then $p_1^1(S)$ is a conditional plan satisfying that $D \models_I G$* **after** *$p_1^1(S)$.*

Theorem 1 shows the soundness of $\pi(\mathcal{P})$. The next theorem shows that $\pi(\mathcal{P})$ is complete in the sense that it can generate all conditional plans which are solutions to $\mathcal{P}$.

Theorem 2. *Let $\mathcal{P} = (D,I,G)$ be a planning problem and p be a conditional plan restricted to the syntax defined in Definition 1. If p is a solution to $\mathcal{P}$ then there exists an answer set S of $\pi(\mathcal{P})$ such that $p = p_1^1(S)$.*

Remark 1. **($\pi(\mathcal{P})$ as a conformant planner).** It is worth noticing that we can view $\pi(\mathcal{P})$ as a conformant planner. To see why, let us consider an answer set S of $\pi(\mathcal{P})$ and $p_1^1(S)$. Theorem 1 implies that $p_1^1(S)$ achieves the goal of $\mathcal{P}$ from every possible initial state of the domain. From the construction of $p_1^1(S)$, we know that if S does not contain a branching literal then $p_1^1(S)$ is a sequence of actions, and hence, a conformant plan. Furthermore, the definition of $\pi(\mathcal{P})$ implies that S contains a branching literal only if $level > 1$. Thus, if we set $level = 1$, $\pi(\mathcal{P})$ is indeed a conformant planner.

Remark 2. **(Size of $\pi(\mathcal{P})$).** It is obvious that we can characterize the size of an action theory by the number of fluents, actions, propositions, literals in the goal, and the number of sensed-fluent literals occurring in propositions of the form (4). Let n be the maximal number among these numbers. It is easy to see that the size of $\pi(\mathcal{P})$ is polynomial in n since the size of each group of rules is polynomial in n.

Remark 3. Because the 0-Approximation is incomplete w.r.t. to the full $\mathcal{A}_K$ semantics [24], there are situations in which $\mathcal{P}$ has solutions but $\pi(\mathcal{P})$ cannot find a solution. E.g., for $\mathcal{P}{=}(D,I,G)$ with $D{=}\{$ **causes**$(a,f,\{g\})$, **causes**$(a,f,\{\neg g\})\}$, $I{=}\emptyset$, and $G{=}f$, $p{=}a$ is a plan achieves f from every initial state; this solution cannot be found by $\pi(\mathcal{P})$.

5 Experiments and Discussions

We have tested our program with a number of domains from the literature. We concentrate on the generation of conditional plans in domains *with sensing actions*. In particular, we compare $\pi(\mathcal{P})$ with the system SGP, one of a few planners that are capable of generating both conditional plans and conformant plans in the presence of sensing actions and incomplete information available, from `http://www.cs.washington.edu/ai/sgp.html` in the *bomb in the toilet with sensing actions*, the *cassandra*, and the *illness* domains. To facilitate the testing, we develop a Sicstus program that translates a planning problem $\mathcal{P} = (D, I, G)$ – specified as a Sicstus program – into the program $\pi(\mathcal{P})$ and then use the **smodels** to compute one solution. All experiments were run on a Pentium III, Fujitsu FMV-BIBLO NB9/1000L laptop, 1GHz with 256 Mb RAM and 40 GB Harddisk. The operating environment is GNU/Linux 2.4 20-8. **lparse** version 1.0.13 and **smodels** version 2.27 are used in our computation. LispWorks 4.2.0 is used in running the SGP system. The source code of the translator and the problem in $\mathcal{A}_K$ ontologies are available at `http://www.cs.nmsu.edu/~tson/ASPlan/Sensing`. We detailed the results in Table 1.

Table 1. Comparison with SGP (Time in milliseconds)

Domains/ Problem	SGP				$\pi(\mathcal{P})$							
	1^{st}	2^{nd}	3^{rd}	Avg.	1^{st}		2^{nd}		3^{rd}		Avg.	
					+	S	+	S	+	S	+	S
Cassandra												
a1-prob	140	170	140	150	387	190	358	210	390	190	378	197
a2-prob	40	40	40	40	811	500	810	510	809	460	810	490
a3-prob	50	50	40	47	282	110	284	110	282	130	283	117
a4-prob	300	300	300	300	704	470	705	480	705	490	705	480
a5-prob	50	30	40	40	185	80	219	70	185	80	196	77
a6-prob	NA [3]	NA	NA	NA	7,990	6,860	7,977	6,850	7,971	6,930	7,979	6,880
a7-prob	120	120	120	120	238	80	239	90	239	80	239	83
Bomb												
bt-1sa	1,790	1,630	1,660	1,693	4,491	3,660	4,420	3,510	4,702	3,640	4,538	3,603
bt-2sa	1,760	1,760	1,710	1,743	5,386	4,220	5,349	4,070	5,285	4,080	5,340	4,123
bt-3sa	1,940	1,910	1,900	1,917	5,404	3,950	5,349	4,140	5,369	4,070	5,374	4,053
bt-4sa	2,130	2,150	2,080	2,120	7,387	5,650	7,286	5,530	7,335	5,530	7,336	5,570
Sick												
sick-3-1	20	20	10	17	437	70	459	50	449	60	448	60
sick-3-2	130	130	160	140	810	320	812	320	791	300	804	313
sick-3-3	500	480	550	510	2,740	1,640	2,735	1,790	2,705	1,760	2,727	1,730
sick-3-4	2,630	2,610	2,650	2,630	3,632	2,390	3,652	2,400	3,633	2,460	3,639	2,417
sick-3-5	17,070	17,370	17,690	17,377	3,749	2,510	3,810	2,620	3,613	2,750	3,724	2,627

[2] LispWorks stops with the error "Memory limit exceeded".

For each problem, we record the computation time needed to find the first solution in both systems in three tries. The average time is also reported. Columns with the heading '+' under the header $\pi(\mathcal{P})$ record the total time (**smodels** and **lparse**) needed to find one answer using $\pi(\mathcal{P})$. The other columns record the time reported by **smodels**. As it can be seen, $\pi(\mathcal{P})$ performs better than SGP in some domains and does not do as well in some other domains. We observe that in many domains, where SGP performs better, the initial state was specified using the PDDL 'oneOf' construct (i.e., the initial state is one of the possible given states). This construct provides SGP with extra information. In our encoding, we omit this information whenever the set of sensing actions is sufficient for the planner to acquire it; otherwise, we add an extra sensing action to allow our planner to determine the initial state. When we do so, $\pi(\mathcal{P})$ does better than SGP (e.g., the problem 'a6-prob'). We also observe that in some problems, where SGP performs better, the search space is rather huge with a lot of repetitions of an action. This can be seen in the problems of the 'Bomb' domain. In this problems, there are several alternatives of the sensing action that detects metal in a package. SGP consistently returns the same solution whereas $\pi(\mathcal{P})$ returns different solutions in different runs.

Final Remarks. We present a sound and complete logic programming encoding of the planning problem with sensing actions in the presence of incomplete information. Our encoding shows that model-based approach to planning can be extended to planning with sensing actions and incomplete information. This distinguishes our planner from other model-based planners that do not deal with sensing actions [2,3,6,7,23] . We compare our planner with the system SGP and obtain encouraging results. In the future, we would also like to investigate methods such as use of domain knowledge to speed up the planning process. Furthermore, due to the fact that our planner can be viewed as a conformant planer, we would like to test our planner against other model-based conformant planners as well.

Acknowledgment. We would like to thank Michael Gelfond for his valuable comments on an earlier draft of this paper. We would also like to thank the anonymous reviewers of the paper for their constructive comments which help us to improve the paper in several ways.

References

1. C. Baral, V. Kreinovich, and R. Trejo. Planning and approximate planning in presence of incompleteness. In *IJCAI*, pages 948–953, 1999.
2. B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *AIPS*, 1998.
3. A. Cimatti and M. Roveri. Conformant planning via model checking. In *ECP*, 21–34, 1999.
4. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of European conference on Planning*, pages 169–181, 1997.
5. T. Eiter et al. The KR System dlv: Progress Report, Comparisons and Benchmarks. *KR*'98.
6. T. Eiter et al. Planning under incomplete information. In CL'2000.
7. T. Eiter et al. A Logic Programming Approach to Knowledge State Planning, II: The DLV$^{\mathcal{K}}$ System. Technical Report, TU Wien, 2003.
8. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.

9. K. Golden. Leap Before You Look: Information Gathering in the PUCCINI planner. In *Proc. of the 4th Int. Conf. on Artificial Intelligence Planning and Scheduling Systems*, 1998.
10. K. Golden, O. Etzioni, and D. Weld. Planning with execution and incomplete informations. Technical report, University of Washington, TR96-01-09, February 1996.
11. K. Golden and D. Weld. Representing sensing actions: the middle ground revisited. In *KR 96*, pages 174–185, 1996.
12. H. Levesque. What is planning in the presence of sensing? In *AAAI*, 1139–1146, 1996.
13. V. Lifschitz. Answer set planning. In *ICLP*, 23–37, 1999.
14. J. Lobo. COPLAS: a COnditional PLAnner with Sensing actions. FS-98-02, AAAI, 1998.
15. J. Lobo, S. Taylor, and G. Mendez. Adding knowledge to the action description language $\mathcal{A}$. In *AAAI 97*, pages 454–459, 1997.
16. R. Moore. A formal theory of knowledge and action. In J. Hobbs and R. Moore, editors, *Formal theories of the commonsense world*. Ablex, Norwood, NJ, 1985.
17. I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. ICLP & LPNMR*, pages 420–429, 1997.
18. I. Niemelä, P. Simons, and T. Soininen. Stable model semantics for weight constraint rules. In *LPNMR*, pages 315–332, 1999.
19. M.A. Peot and D.E. Smith. Conditional Nonlinear Planning. In *AIPS*, pages 189–197, 1992.
20. L. Pryor and G. Collins. Planning for contingencies: A decision-based approach, *JAIR*, 1996.
21. J. Rintanen. Constructing conditional plans by a theorem prover. *JAIR*, 10:323–352, 2000.
22. R. Scherl and H. Levesque. The frame problem and knowledge producing actions. AAAI'96.
23. D.E. Smith and D.S. Weld. Conformant Graphplan. In *AAAI*, pages 889–896, 1998.
24. T.C. Son and C. Baral. Formalizing sensing actions - a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, January 2001.
25. M. Thielscher. Programming of Reasoning and Planning Agents with FLUX. KR'02, 2002.
26. M. Thielscher. Representing the knowledge of a robot. In *KR'00*, 109–120, 2000.
27. D. Weld, C. Anderson, and D. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proceedings of AAAI 98*, 1998.

Deduction in Ontologies via ASP

Terrance Swift

Dept. of Computer Science, SUNY Stony Brook
tswift@cs.sunysb.edu

Abstract. Ontologies have become an important methodology for representing knowledge, particularly for allowing agents to interchange knowledge over the world-wide-web. From an abstract point of view, an ontology can be seen as a theory about a set of classes. The language underlying the ontology may or may not be decidable; if it is, it is often called a *description logic*, and the problem of determining whether one description logic formula implies (or *subsumes*) another is fundamental to deduction in ontologies. This paper models description logics as *first-order* theories, and employs model-theoretic techniques to determine properties of various description logics. These properties are used to design efficient engines to generate Answer Set Programs that perform deduction in ontologies. This approach contrasts to tableaux theorem proving techniques that are more commonly used. The resulting system serves as an experimental platform to explore the combination of logic-programming based techniques for non-monotonic reasoning and constraint handling with description-logic based deduction. Specifically, we use ASP to create a small but powerful theorem prover for the description logic $\mathcal{ALCQI}$. While $\mathcal{ALCQI}$ is *P-space* complete, our deduction engine requires exponential space in the worst case. However experiments show that its time is roughly comparable to the one of the best tableaux-based engined, DLP [1], even though DLP is written for a simpler description logic, $\mathcal{ALCN}$ [1].

1 Introduction

Ontologies have many strengths as a knowledge representation technique. First, ontologies can be formalized so that they have a clear semantics. Ontologies consist of *classes* and *objects* that belong to these classes. Classes are described as formulas (traditionally called *class expressions*) in an underlying logic, which also models subclass and other binary relations between the classes, inheritance, and other features. If the logic underlying an ontology is decidable, it is sometimes called a description logic, and for the purposes of this paper, we assume a decidable logic underlies an ontology. As a second strength, when an ontology is based on a description logic, the complexity of various decision problems have been extensively studied (see e.g. [2]) particularly for determining the consistency of a set of formulas. Efficient algorithms also exist to check consistency [3,4] and

[1] The full version of this paper, containing all proofs, additional examples and a more detailed exposition of concepts can be obtained at www.cs.sunysb.edu/~tswift.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 275–288, 2004.

have been implemented, usually via tableaux-based theorem provers (e.g. [5,1]. As a final strength, many aspects of ontologies have an intuitive appeal. Much of the structure of an ontology, such as the implication relation between classes, the relation of classes and objects, and the relations of classes through binary relations can be graphically visualized and manipulated through an graphical ontology editor such as Protege (`protege.stanford.edu`).

In this paper, we explore how deduction in description logics can be implemented via *Answer Set Programming (ASP)* [2]. Specifically,

- We describe a transformation from a class expression ϕ in the description logic $\mathcal{ALCQI}$ into a formula in first order logic, prove it sound and complete, provide an upper bound on the size of the smallest model for ϕ, and define and prove graph-theoretic properties of these models. Not all of these results are new (cf. [7]), but the proofs are model-theoretic in nature, unlike those that are usual in the description logic literature.
- We show how ϕ can be transformed into an answer set program, using an optimization we call Magic ASP. Because Quantified Boolean Formulas can be polynomially transformed to $\mathcal{ALCQI}$ formulas but such a transformation to answer set programs is not known, this transformation will be exponential in the worst case. However, conditions in which worst-case behavior are attained appear unrealistic for practical ontologies.
- A deduction engine is implemented in a relatively short (1400 lines) logic program, where the transformation is written in XSB (`xsb.sourceforge.net`) and the resulting answer set program sent to either SModels [8] or DLV [9].
- The resulting engine is compared to a tableaux-based engine, DLP [1]. Although the DLP engine does not implement qualified number restrictions or inverse relations, times for the XSB/SModels deduction engine are roughly comparable to DLP for some sample benchmarks.

2 First-Order Semantics for Description Logics

A language $\mathcal{L}_D$ for a description logic contains a set $\mathcal{L}_D^{\mathcal{A}_C}$ of names for atomic classes, a distinct set $\mathcal{L}_D^{\mathcal{A}_R}$ of names for atomic relations, along with names for the natural numbers. Note that $\mathcal{L}_D$ contains no variables. Formulas in $\mathcal{L}_D$ are termed *class expressions*, defined inductively as follows. A *relation* in $\mathcal{L}_D$ is either an atomic relation name in $\mathcal{L}_D^{\mathcal{A}_R}$ or R^-, where R is a relation. A *class expression*, C, is either an atomic class name in $\mathcal{L}_D^{\mathcal{A}_C}$ or is formed by one of the following constructions in which a is a primitive class name, C_1 and C_2 class expressions, R a relation, and n a natural number.

$$C \leftarrow a \mid \top \mid \neg C_1 \mid C_1 \sqcap C_2 \mid C_1 \sqcup C_2 \mid all(R, C_1) \mid exists(R, C_1) \\ \mid atLeast(n, R, C_1) \mid atMost(n, R, C_1)$$

The constructions $atLeast(n, R, C_1)$ and $atMost(n, R, C_1)$ are called *qualified number restrictions*. Structures for $\mathcal{L}_D$ have a special form.

[2] A somewhat different translation has been proposed independantly in [6].

Definition 1. *Let $\mathcal{L}_D$ be a description logic language. A structure $< \Delta, \mathcal{I} >$ for $\mathcal{L}_D$ maps atomic class and relation names to a set Δ such that: for $a \in \mathcal{L}_D^{\mathcal{A}_C}, a^{\mathcal{I}} \subseteq \Delta$; for $r \in \mathcal{L}_D^{\mathcal{A}_R}, r^{\mathcal{I}} \subseteq \Delta \times \Delta$*

Definition 2. *Let $\mathcal{L}_D$ be a description logic language, n a natural number, C_1 and C_2 class expressions over $\mathcal{L}_D$, R a relation over $\mathcal{L}_D$, and $< \Delta, \mathcal{I} >$ a structure for $\mathcal{L}_D$. $< \Delta, \mathcal{I} >$ is extended to class expressions and relations as follows.*

$$
\begin{aligned}
[t]\ \top^{\mathcal{I}} &= \Delta & (\neg C_1)^{\mathcal{I}} &= \Delta - C_1^{\mathcal{I}} \\
(C_1 \sqcap C_2)^{\mathcal{I}} &= C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} & (C_1 \sqcup C_2)^{\mathcal{I}} &= C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}} \\
(exists(R, C_1))^{\mathcal{I}} &= \{x \in \Delta | \exists y. \langle x, y \rangle \in R^{\mathcal{I}} \wedge y \in C_1^{\mathcal{I}}\} \\
(all(R, C_1))^{\mathcal{I}} &= \{x \in \Delta | \forall y. \langle x, y \rangle \in R^{\mathcal{I}} \rightarrow y \in C_1^{\mathcal{I}}\} \\
(atLeast(n, R, C_1))^{\mathcal{I}} &= \{x \in \Delta | \sharp\{y | \langle x, y \rangle \in R^{\mathcal{I}} \wedge y \in C_1^{\mathcal{I}}\} \geq n\} \\
(atMost(n, R, C_1))^{\mathcal{I}} &= \{x \in \Delta | \sharp\{y | \langle x, y \rangle \in R^{\mathcal{I}} \wedge y \in C_1^{\mathcal{I}}\} \leq n\} \\
(R^{-})^{\mathcal{I}} &= \{(o', o) \in \Delta \times \Delta | (o, o') \in R^{\mathcal{I}}\}
\end{aligned}
$$

where for any set S, $\sharp S$ is the cardinality of S. $< \Delta, \mathcal{I} >$ is a model of a class expression C if $C^{\mathcal{I}}$ is non-empty.

If only primitive relations are allowed, and there are no qualified number restrictions the description logic is called $\mathcal{ALC}$ [3]. The presence of inverse relations is designated by the letter $\mathcal{I}$ and the presence of qualified number restrictions by the letter $\mathcal{Q}$, so that the description logic discussed here may be termed $\mathcal{ALCQI}$ (see e.g. [2] or [10] for more on description logic terminology).

Since a structure for a description logic is function-free (apart from constants) and contains only unary and binary predicates, it can be represented graphically.

Definition 3. *Let $< \Delta, \mathcal{I} >$ be a structure for a description logic language $\mathcal{L}_D$. The structure graph of $< \Delta, \mathcal{I} >$ is a directed graph whose vertexes are the elements of Δ and such that there is an edge labeled r_i between two vertexes d_i and d_j, iff $(d_i, d_j) \in r_i^{\mathcal{I}}$. In addition, each vertex d is labeled with any atomic class name a for which $d \in a^{\mathcal{I}}$.*

Example 1. Let $\mathcal{L}_D^{\mathcal{A}_C} = \{a_0, a_1\}$, $\mathcal{L}_D^{\mathcal{A}_R} = r_0$. Figure 1 depicts the graph for the structure $< \Delta_1, \mathcal{I}_1 >$ where $\Delta_1 = \{d_0, d_1, d_2, d_3\}; a_0^{\mathcal{I}_1} = \{d_1, d_2\}; a_1^{\mathcal{I}_1} = \{d_1\}; r_0^{\mathcal{I}_1} = \{(d_0, d_1), (d_0, d_2), (d_2, d_3)\}$. Consider the class expression: $C_1 = (exists(r_0, a_1) \sqcap exists(r_0, exists(r_0, \neg a_1))) \sqcap all(r_0, a_0)$ Verifying that $C_1^{\mathcal{I}_1} = \{d_0\}$ is a straightforward application of Definition 2, so that $< \Delta_1, \mathcal{I}_1 >$ is a model of C_1. C_1 is in $\mathcal{ALC}$ since it has no occurrences of qualified number restrictions or inverse relations.

Definitions 1 and 2 clearly define a set-theoretic semantics of class expressions, and are traditionally used by the description logic community (see, e.g. [10]). However, the semantics of class expressions can also be defined by a transformation into formulas of first-order logic, so that the set-theoretic semantics can be reconstructed and extended using the model theory of first-order logic. Let $\mathcal{L}_D$ be a description-logic language. $\mathcal{L}_D$ is extended to a *first-order description logic language*, $\mathcal{L}'_D$ in the following manner. The constants of $\mathcal{L}'_D$ consist

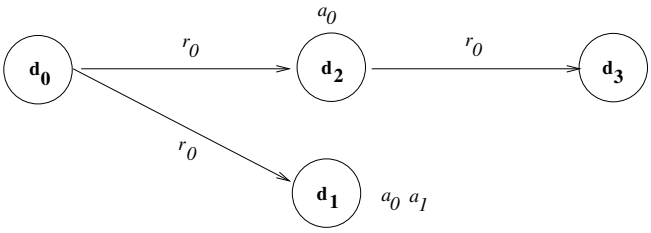


Fig. 1. A Structure Graph of $< \Delta_1, \mathcal{I}_1 >$

of $\mathcal{L}_D^{\mathcal{A}_R}$, and $\mathcal{L}_D^{\mathcal{A}_C}$; and the predicates of $\mathcal{L}'_D$ include equality (=), $elt/2$, $rel/3$, and $univ/1$. $\mathcal{L}'_D$ also contains a sequence of variable names, $\mathcal{L}_D^X$ which we assume to be countable and unique. The following definition associates a first-order sentence in $\mathcal{L}'_D$ with a class expression in $\mathcal{L}_D$.

Definition 4. *Let C a class expression formed over a description logic language, $\mathcal{L}_D$. Let $\mathcal{L}'_D$ be the first-order extension of $\mathcal{L}_D$, and let the N^{th} element of $\mathcal{L}_D^X$ be denoted by X_N. Then the first-order counterpart of C, $\tau(C)$, is a sentence $\exists X_0.\tau(C, X_0)$ over $\mathcal{L}'_D$. $\tau(C, X_0)$ itself is defined by the following rules, which use a global variable number, v that is initialized to 1 at the start of the transformation.*

- *if $C = a \in \mathcal{L}_D^{\mathcal{A}_C}$, then $\tau(C, X_N) = elt(X_N, a)$.*
- *if $C = \top$, then $\tau(C, X_N) = univ(X_N)$.*
- *if $C = (C_1 \sqcap C_2)$, then $\tau(C, X_N) = \tau(C_1, X_N) \wedge \tau(C_2, X_N)$.*
- *if $C = (C_1 \sqcup C_2)$, then $\tau(C, X_N) = \tau(C_1, X_N) \vee \tau(C_2, X_N)$.*
- *if $C = \neg C_1$, then $\tau(C, X_N) = \neg\tau(C_1, X_N)$.*
- *if $C = exists(\rho_0, C_1)$, then $\tau(C, X_N) = \exists X_v.\tau_\rho(\rho_0, X_N, X_v) \wedge \tau(C_1, X_v)$, and where v is incremented before expanding τ again.*
- *if $C = all(\rho_0, C_1)$, then $\tau(C, X_N) = \neg\exists X_v.(\tau_\rho(\rho_0, X_N, X_v) \wedge \tau(\neg C, X_v))$. and where v is incremented before expanding τ again.*
- *if $C = atLeast(m, \rho_0, C_1)$, then $\tau(C, X_N) =$*

  $$\exists X_v, \ldots, \exists X_{v+m-1}[\\ (\tau_\rho(\rho_0, X_N, X_v) \wedge \tau(C_1, X_v)) \wedge \ldots \wedge (\tau_\rho(\rho_0, X_N, X_{v+m-1}) \wedge \tau(C_1, X_{v+m-1}) \\ \wedge X_i \neq X_{i+1} \wedge \ldots \wedge X_i \neq X_{v+m-1} \wedge X_{i+1} \neq X_{i+2} \wedge \ldots \wedge X_{i+1} \neq X_{v+m-1}]$$

  and where v is incremented m times before expanding τ again.
- *if $C = atMost(m, \rho_0, C_1)$, then $\tau(C, X_N) = \tau(\neg atLeast(m+1, \rho_0, C_1), X_N)$*

$\tau_\rho(\rho', X_N, X_M)$ is defined as follows

- *if $\rho' = r \in \mathcal{L}_D^{\mathcal{A}_R}$, $\tau_\rho(\rho_i, X_N, X_M) = rel(X_N, r, X_M)$*
- *if $\rho' = \rho''^-$, $\tau_\rho(\rho', X_N, X_M) = \tau(\rho'', X_M, X_N)$*

Definition 5. *Let $< \Delta, \mathcal{I} >$ be a structure for a description logic language $\mathcal{L}_D$. We assume, without loss of generality, that Δ is distinct from $\mathcal{L}^{\mathcal{A}_C}$ and $\mathcal{L}^{\mathcal{A}_R}$. Then a structure $< \Delta \cup \mathcal{L}^{\mathcal{A}_C} \cup \mathcal{L}^{\mathcal{A}_R}, \mathcal{J} >$ for $\mathcal{L}'_D$* induced by *$< \Delta, \mathcal{I} >$ is defined as follows.*

- $a^{\mathcal{J}} = a$ *iff* $a \in \mathcal{L}^{\mathcal{A}_C}$ *or* $a \in \mathcal{L}^{\mathcal{A}_R}$;
- $(s_1) \in univ^{\mathcal{J}}$ *iff* $s_1 \in \Delta$
- $(s_1, s_2) \in elt^{\mathcal{J}}$ *iff* $s_1 \in \Delta$, $s_2 \in \mathcal{L}^{\mathcal{A}_C}$, *and* $s_1 \in s_2^{\mathcal{I}}$
- $(s_1, s_2, s_3) \in rel^{\mathcal{J}}$ *iff* $s_1, s_3 \in \Delta$, $s_2 \in \mathcal{L}^{\mathcal{A}_R}$, *and* $(s_1, s_3) \in s_2^{\mathcal{I}}$

Elements of $\mathcal{L}_D^{\mathcal{A}_R}$, and $\mathcal{L}_D^{\mathcal{A}_C}$ are mapped to themselves in order to prevent interpretations in which a class name is a member of a class name or a relation is a constituent of another relation. The full version of this paper shows that a first-order structure induced by a description logic interpretation is unique.

Example 2. Let $\mathcal{L}_D$ be the language from Example 1. Consider the structure $\Delta_1 = \{d_0, d_1, d_2\}; a_0^{\mathcal{I}_2} = \{d_1\}; a_1^{\mathcal{I}_2} = \{d_1\}; r_0^{\mathcal{I}} = \{(d_0, d_1), (d_1, d_2)\}$, and class expression: $C_2 = exists(r_0, a_1) \sqcap all(r_0, a_0) \sqcap exists(r_0, exists(r_0, \neg a_1)) \sqcap atMost(1, r_0, a0)$. C_2 is in $\mathcal{ALCQ}$ since it has a qualified number restriction $(atMost(1, r_0, a_0))$. Note that $< \Delta_1, \mathcal{I}_2 >$ is a model of C_2, but that $< \Delta_1, \mathcal{I}_1 >$ from Example 1 is not. The first order counterpart, $\tau(C_2)$, of C_2 is:

$$
\begin{aligned}
&\exists X_0, X_1[rel(X_0, r_0, X_1) \wedge elt(X_1, a_1)] \\
\wedge &\neg \exists X_2[rel(X_0, r_0, X_2) \wedge \neg elt(X_2, a_0)] \\
\wedge &\ \exists X_3[rel(X_0, r_0, X_3) \wedge \exists X_4[rel(X_3, r_0, X_4) \wedge \neg elt(X_4, a_1)]] \\
\wedge &\neg \exists X_5, X_6[rel(X_0, r_0, X_5) \wedge elt(X_5, a_0) \wedge rel(X_0, r_0, X_6) \wedge elt(X_6, a_0) \wedge X_5 \neq X_6]
\end{aligned}
$$

and $\tau(C_2)$ is satisfiable in the structure induced by $< \Delta_1, \mathcal{I}_2 >$.

Theorem 1. *Let C be a class expression formed over a description logic language $\mathcal{L}_D$. Then $< \Delta, \mathcal{I} >$ is a model of C iff the structure $< \Delta \cup \mathcal{L}^{\mathcal{A}_C} \cup \mathcal{L}^{\mathcal{A}_R}, \mathcal{J} >$ induced by $< \Delta, \mathcal{I} >$ is a model of $\tau(C)$.*

2.1 Properties of Description Logic Models

The next two theorems explore properties of description logics that are relevant to the algorithms of Section 3. The following theorem gives an upper bound for the size of the smallest model for $\tau(C)$. This bound will be used by the algorithms described in the next section. Before stating the theorem, we introduce some terminology. Let C be a class expression. Clearly, each variable X_i in $\tau(C)$ is quantified a single time, always by an existential quantifier. We define $level(X_i)$ as the number of negation symbols encountered in a path from the root of $\tau(C)$ to $\exists X_i$. The number of variables with even level in $\tau(C)$ is called the *variable space* of $\tau(C)$.

Theorem 2. *Let C be a consistent class expression over a description logic language $\mathcal{L}_D$. Then C has a model $< \Delta', \mathcal{I} >$ of C such that $|\Delta'| \leq V$, where V is the variable space of $\tau(C)$.*

Properties of structure graphs can be related to the types of expressions they model. More specifically, structure graphs can be extended to first-order structures and the following theorem proved.

Theorem 3. *1. Let C be an $\mathcal{ALC}$ class expression. If $\tau(C)$ has a model, $\tau(C)$ has a model whose structure graph is a rooted tree.*
2. Let C be an $\mathcal{ALCI}$ class expression. If $\tau(C)$ has a model, $\tau(C)$ has a model whose structure graph is a tree.
3. Let C be an $\mathcal{ALCQI}$ class expression. If C has a model, $\tau(C)$ has a model whose structure graph is acyclic.

3 From First-Order Formulas to Answer Set Programs

In this section we assume a familiarity with the terminology of Stable Model Semantics and Answer Set Programming, including weight constraints (see e.g. [11]). For simplicity of presentation, we present unoptimized ASP programs; optimizations will be presented in Section 4.

3.1 Consistency Checking of $\mathcal{ALC}$ Class Expressions

The main idea behind using ASP to check consistency of a class expression C is to translate C into its first-order counterpart $\tau(C)$ via Definition 4, and then to translate $\tau(C)$ into an ASP program $P_{\tau(C)}$ using techniques described below. One of the rule heads in $P_{\tau(C)}$ is an atom `sat0` such that $P_{\tau(C)}$ will have a stable model, M containing `sat0` if and only if the $elt/2$ and $rel/3$ atoms true in M form a model (in the sense of Definition 5) of $\tau(C)$. The basic action of the ASP program, $P_{\tau(C)}$ is to produce various sets of ground instances of $elt/2$ and $rel/3$ predicates, and then check whether each set can be expanded into a (consistent) stable model. The $rel/3$ and $elt/2$ atoms that occur in $P_{\tau(C)}$ are called *structure elements* of $P_{\tau(C)}$; those that occur as facts in $P_{\tau(C)}$ are called *deterministic*, others are *non-deterministic*. In examples below, we adopt the convention that non-deterministic structure elements are starred. The set of all structure elements is called the *structure space* of $P_{\tau(C)}$. Generation of non-deterministic structure elements can be programmed in several ways depending on what is most efficient for a given ASP system.

Example 3. Consider the $\mathcal{ALC}$ class expression C_1 from Example 1. Using Theorem 2, $\tau(C_1)$ can be ground to the sentence.

$$\begin{aligned} & rel(d_0, r_0, d_1) \wedge elt(d_1, a_1) \\ \wedge & rel(d_0, r_0, d_2) \wedge rel(d_2, r_0, d_3) \wedge \neg elt(d_3, a_1) \\ \wedge & (\neg rel(d_0, r_0, d_1) \vee (rel(d_0, r_0, d_1) \wedge elt(d_1, a_0))) \\ & \wedge (\neg rel(d_0, r_0, d_2) \vee (rel(d_0, r_0, d_1) \wedge elt(d_1, a_0))) \end{aligned}$$

A schematic ASP program, $P_{\tau(C_1)}$ is shown in Figure 2.

The atom `sat0` is true in the stable model in which {`rel*(d0,r1,d1), elt*(d1,a1), rel*(d0,r1,d2), rel*(d2,r1,d3), elt*(d1,a0), elt*(d2,a0)`} are all true. This stable model corresponds to the model graph in Figure 1. We will show below that the actual ASP theorem prover can generate much more efficient code than indicated by this schematic example.

```
sat0:- rel*(d0,r0,d1), elt*(d1,a1),
  rel*(d0,r0,d2), rel*(d2,r0,d3), not elt*(d3,a1), sat1, sat2.

sat1:-  not rel*(d0,r0,d1).    sat1:- rel*(d0,r0,d1),elt*(d1,a0).

sat2:-  not rel*(d0,r0,d2).    sat2:- rel*(d0,r0,d2),elt*(d2,a0).
```

Fig. 2. A schematic ASP program for C_1 of Example 1

It is not hard to see how the structure of $P_{\tau(C_1)}$ in Example 3, reflects the first-order counterpart, $\tau(C_1)$, of C_1. At the same time, it is not obvious that if $P_{\tau(C_1)}$ does not have a stable model, than C_1 is inconsistent — perhaps searching a larger structure space would lead to a model of C_1. We define an algorithm that transforms $\tau(C)$ to a variable-free formula, ground(C) which maps easily to $P_{\tau(C)}$. We show that if C has a model, it will have a model whose true $rel/3$ and $elt/2$ facts are subsets of the structure space of $P_{\tau(C)}$. ground(C) is shown in Figure 3.1 for $\mathcal{ALC}$ class expressions. The algorithm works on class expressions in *negation reduced form*: any class expression C can be rewritten into a class expression negation reduced form by pushing negation down to atomic class names. ground(C) uses the notion of levels of a variable in $\tau(C)$ (introduced in Section 2.1), and iterates through these levels, rewriting $\tau(C)$ to eliminate variables, and adding new atoms to the structure space. The level of a subformula S in $\tau(C)$ is defined as the minimal level of any variable in S, or 0 if S has no variables. Its further actions are described fully in the full version of this paper.

/* C is a class expression in negation reduced form */
$ground(C) = \tau(C); cur\_lev = 0; cur\_struct = \emptyset$
For each variable X in $ground(C)$, set $level(X)$ to its level in $ground(C)$
repeat
  For each subformula $S = \exists X_j[rel(d_i, r_k, X_j) \wedge \tau(\neg C, X_j)]$ with level cur_lev
    If cur_lev is even
      Rewrite S with a new constant d' in the place of X_i, removing the quantifier
    If cur_lev is odd
      If there exists a d_m such that $rel(d_i, r_k, d_m) \in cur\_struct$
        Replace S in $ground(C)$ by
        $\bigwedge_{rel(d_i,r_k,d_m)\in cur\_struct} \neg rel(d_i, r_k, d_m) \vee (rel(d_i, r_k, d_m) \wedge \tau(C, d_m))$
      Otherwise replace S by $true$.
  Set cur_struct to the set of $rel/3$ or $elt/2$ atoms occurring positively in $ground(C)$
  cur_lev++; Remove leading negations for all subformulas of level cur_lev;
until no variables are left in $ground(C)$

Fig. 3. Algorithm GROUND(C) for $\mathcal{ALC}$ Expressions

Theorem 4. *Let C be an $\mathcal{ALC}$ class expression in negation reduced form. Then $\tau(C)$ has a model, iff ground(C) has a model.*

Example 4. Continuing from Example 3, it can be seen that $ground(C_1)$ is made into the program of Figure 2 simply by creating an initial rule (`sat0`), whose body contains each conjunct in $\mathcal{F}_P$. For each conjunct with an immediate subformula that is not a literal, such as $(\neg rel(d_0, r_0, d_1) \vee (rel(d_0, r_0, d_1) \wedge elt(d_1, a_0)))$ a new rule is generated with a new head (`sat1` or `sat2`) and whose body is a recursive translation of the subformulas.

3.2 Magic ASP

Figure 2 depicts a direct translation of ground(C_1) into an ASP program. However, such a direct translation can be optimized in several ways: first, it can be noted that since $rel(d_0, r_0, d_1)$ occurs directly as a conjunct in $ground(C_1)$, then if C_1 is to have a model generated from the structure space of $P_{\tau(C_1)}$, $rel(d_0, r_0, d_1)$ must be true in that model. As a result, Figure 2 could be rewritten so that $rel(d_0, r_0, d_1)$ and other such atoms are deterministic. (this would be indicated by removing the stars from these atoms in Figure 2). Furthermore, it is not hard to see that in other cases, when several literals occur in a disjunction, it is more efficient to determine their truth values together. Efficient translation of a propositional formula $\mathcal{F}$ in negation reduced form to an ASP program is done by the algorithm magic_asp($\mathcal{F}$) (Figure 3.2), which allows ASP evaluation to mimic a top-down evaluation of the formula produced by $ground(C)$. magic_asp($\mathcal{F}$) can best be described through an example.

```
magic_asp(F)
    magic_asp1(F,true,true,true)

magic_asp1(F,PGuard,FGuard,EGuard)
/* F is a prop. formula in negation reduced form; PGuard the parent guard;
       FGuard is the guard for this conjunct; EGuard is exclusion guard */
    if EGuard = true /* FGuard and PGuard are also true */
       For each conjunct C in F that is a literal
          emit(C)
       For each conjunct, D1 ∨ D2 in F that is not a literal
          G1 =newGuardAtom(); G2 =newGuardAtom();
          magic_asp1(D1,FGuard,G1,G2); magic_asp1(D2,FGuard,G2,G1);
    else /* EGuard is a guard atom */
       emit(FGuard :- PGuard,not EGuard);
       For each conjunct C in F that is a literal
          emit(C :- FGuard)
       For each conjunct D1 ∨ D2 in F that is not a literal
          G1 =newGuardAtom(); G2 =newGuardAtom();
          magic_asp1(D1,FGuard,G1,G2); magic_asp1(D2,FGuard,G2,G1);
```

Fig. 4. The Magic Asp Algorithm for $\mathcal{ALC}$ Expressions

Example 5. Consider the actions of magic_asp($\mathcal{F}$) on

$$C_5 = a_0 \sqcap (exists(r_0, (a_1 \sqcup a_2)) \sqcup exists(r_1, (\neg a_1 \sqcup \neg a_2))).$$

```
rel(d0,r0,d1):- guard0*,not(guard0a*).  %exists(r0,(a1 ⊔ a2))

rel(d0,r1,d2):- guard0a*,not(guard0*).  %exists(r1,(¬a1 ⊔ ¬a2))

guard1:- guard0*,not guard1a.           guard1a:- guard0*,not guard1.

elt(d1,a1):- guard1.                    %exists(r0,(a1 ⊔ a2))
elt(d1,a2):- guard1a.                   %exists(r0,(a1 ⊔ a2))

guard2:- guard0a*,not guard2a.          guard2a:- guard0a*,not guard2.

-elt(d1,a1):- guard1.                   %exists(r1,(¬a1 ⊔ ¬a2))
-elt(d1,a2):- guard1a.                  %exists(r1,(¬a1 ⊔ ¬a2))

elt(d0,a0)
```

Fig. 5. Magic ASP program for C_5

ground(C_5) produces the formula:

$$elt(d_0, a_0) \wedge$$
$$((rel(d_0, r_0, d_1) \wedge (elt(d_1, a_1) \vee elt(d_1, a_2)))$$
$$\vee (rel(d_0, r_0, d_2) \wedge (-elt(d_2, a_1) \vee -elt(d_1, a_2))))$$

The output of magic_asp(ground(C_5)) is shown in Figure 5. The routine immediately calls magic_asp1(ground(C_5),true,null,null). The first conjunct in $ground(C_5)$ is the literal $elt(d_0, , a_0)$ which is output to $P_{\tau(C_5)}$ as a fact. The second conjunct, is not a literal, so for each each disjunct, new guard atoms (guard0 and guard0a) are created and magic_asp1 is called recursively. The action for the first disjunct is to call magic_asp1($(rel(d_0, r_0, d_1) \wedge (elt(d_1, a_1) \vee elt(d_1, a_2)))$,true,guard0,guard0a). On this call, the rule `guard0 :- not guard0a` is output, ensuring that either `rel(d0,r0,d1)` or `rel(d0,r0,d2)` is generated, but not both. Next, the conjuncts of the formula are checked. The first conjunct is a literal, so that the rule `rel(d0,r0,d1) :- guard0` is output, while the second conjunct is not, so that new guards are generated and the algorithm once again called recursively. For the call magic_asp1($elt(d_1, a_1)$,guard0,guard1,guard1a), the guard rule `guard1:- guard0,not guard1a` is output, ensuring that `elt(d1,a1)` is not be generated unless `rel(d0,r0,d1)` is also generated. Finally, in Figure 5 the literals $-elt(d_2, a_1)$ and $-elt(d_2, a_2)$ from $ground(C_5)$ lead to the generation of explicitly negated objective literals in the magic ASP program.

In the following theorem, a satisfying assignment of a propositional formula $\mathcal{F}$ is a set A of atoms in $\mathcal{F}$ such that if every atom in A is true, then $\mathcal{F}$ is true.

Theorem 5. *Let $\mathcal{F}$ be a propositional formula in negation reduced form, and let P be the program produced by* magic_asp(F)*. Then S is a minimal satisfying assignment of $\mathcal{F}$ iff S is equal to $M|_{\mathcal{F}}$ where $M|_{\mathcal{F}}$ is a stable model of P restricted to the atoms in $\mathcal{F}$.*

3.3 Consistency Checking of $\mathcal{ALCQ}$ Class Expressions

To use ASP to check consistency of $\mathcal{ALCQ}$ class expressions the model construction search must be adapted so that $atMost$ and $atLeast$ constructors are handled properly. This causes two main changes in the grounding algorithm. First, when existential quantifiers are grounded, they must be grounded not only with new constants, but also with certain older constants. Second, both $atMost$ and $atLeast$ constructors are compiled into weight constraints. This section makes use of the definition

$$(\exists n X_i)[\phi] =_{def} \exists X_i, \ldots, X_{i+n}[\phi \ldots \wedge \phi(X_{i+1}/X_i) \\ \wedge X_i \neq X_{i+1} \ldots \wedge X_i \neq X_{i+n} \ldots \wedge X_{i+n-1} \neq X_{i+n}]$$

which is used to construct the first-order counterpart of the $atLeast$ constructor (Definition 4). In addition, for positive integers m, n and sentences σ_i the *weight constraint*: $m\ \{\sigma_i, ..., \sigma_n\}\ m$ is satisfied in a model $\mathcal{M}$ if $\mathcal{M}$ satisfies at least m and at most n of the σ_i. We now turn to an example that illustrates the details of our grounding algorithm. For purposes of presentation, we omit the use of Magic ASP in this section.

Example 6. Consider again the class expression C_2 from Example 2: Using the new notation, its first-order counterpart, $\tau(C_2)$, is

$$\begin{array}{l} \quad \exists X_0, X_1[rel(X_0, r_0, X_1) \wedge elt(X_1, a_1)] \wedge \neg \exists X_2[rel(X_0, r_0, X_2) \wedge \neg elt(X_2, a_0)] \\ \wedge \quad \exists X_3[rel(X_0, r_0, X_3) \wedge \exists 1 X_4[rel(X_3, r_0, X_4) \wedge \neg elt(X_4, a_1)]] \\ \wedge \neg \exists 2 X_5[rel(X_0, r_0, X_5) \wedge elt(X_5, a_0)] \end{array}$$

Next, $ground(C_2)$ is produced

$$\begin{array}{l} \quad rel(d_0, r_0, d_1) \wedge elt(d_1, a_1) \\ \wedge (\neg rel(d_0, r_0, d_1) \vee (rel(d_0, r_0, d_1) \wedge elt(d_1, a_0))) \\ \wedge ((rel(d_0, r_0, d_3) \wedge rel(d_3, r_0, d_4) \wedge \neg elt(d_4, a_1)) \\ \qquad \vee (rel(d_0, r_0, d_1) \wedge rel(d_1, r_0, d_4) \wedge \neg elt(d_4, a_1))) \\ \wedge \neg 2\{(rel(d_0, r_0, d_1) \wedge elt(d_1, a_0)), (rel(d_0, r_0, d_3) \wedge elt(d_3, a_0))\} \end{array}$$

and then translated to $P_{\tau(C_2)}$ in Figure 6.

```
sat0:- rel*(d0,r1,d1), elt*(d1,a1), sat2, sat3, sat4,
  {rel(d0,r1,d1),rel(d0,r1,d3)} 1.

sat2:- rel*(d0,r1,d4),                sat2:- rel*(d0,r1,d3),
  rel*(d1,r1,d4),                       rel*(d3,r1,d4),
  not elt*(d4,a1).                      not elt*(d4,a1).

sat3:- not rel*(d0,r0,d1).            sat3:- rel*(d0,r0,d1),elt*(d1,a0).

sat4:- not rel*(d0,r0,d3).            sat4:- rel*(d0,r0,d3),elt*(d3,a0).
```

Fig. 6. A schematic ASP Program for C_2 of Example 2

For an $\mathcal{ALCQ}$ class expression, ground(C) is given in Figure 7. The first main change from Figure 3.1 occurs for grounding a subformula $S = \exists_n X_j[rel(d_i, r_k, X_j) \wedge \tau(\neg C, X_j)]$ whose level is even. S may need to be ground with $n > 1$ new constants and the grounding placed into a weight constraint. Also, S must be grounded with any d_j such that $rel(d_i, r_k, X_j) \in cur\_struct$, since relations in the structure may have to be "reused" so that formulas arising from the $atMost$ constructor can be satisfied. Similarly, if the level of S is odd, it may need to be ground with $n > 1$ new constants to produce a weight constraint. These arguments can be formalized to prove:

```
/* C is a class expression in negation reduced form */
1   ground(C) = τ(C); cur_lev = 0; cur_struct = ∅
    For each variable X in ground(C), set level(X) to its level in ground(C)
    repeat
      For each subformula S = ∃_n X_j[rel(d_i, r_k, X_j) ∧ τ(¬C, X_j)] with level cur_lev
5       If cur_lev is even
          Let new be a set of n new constants
          Let Set be the set of formulas (rel(d_i, r_k, d_m) ∧ τ(C, d_m))
            such that d_m ∈ new or rel(d_i, r_k, d_m) ∈ cur_struct}
          If (n > 1) Replace S by n {Set} Else Replace S by ⋁ Set
10      If cur_lev is odd
          If there exists a d_m such that rel(d_i, r_k, d_m) ∈ cur_struct
            Let Set be the set of formulas ¬rel(d_i, r_k, d_m) ∨ (rel(d_i, r_k, d_m) ∧ τ(C, d_m))
              such that rel(d_i, r_k, d_m) ∈ cur_struct
            If (n > 1) Replace S by {Set} n Else Replace S by ⋀ Set
15        Else replace S by true.
      Set cur_struct to the set of rel/3 and elt/2 atoms occurring positively in ground(C)
      cur_lev + +; Remove leading negations for all subformulas of level cur_lev;
    until no variables are left in ground(C)
```

Fig. 7. Algorithm GROUND(C) for $\mathcal{ALCQ}$ Expressions

Theorem 6. *Let C be an $\mathcal{ALCQ}$ class expression in negation reduced form. Then $\tau(C)$ has a model, iff ground(C) has a model.*

4 An ASP Theorem Prover for Class Expressions

Based on the algorithms described in Section 3, an ASP theorem prover, *CDFTP*, was written to check consistency of class expressions. *CDFTP* is intended to be used in the Cold Dead Fish Ontology Management System [12], and is available via `xsb.sourceforge.net`. The main steps of CDFTP are as follows.

1. In the analysis phase, properties of C are analyzed for the applicability of optimization, and the structure space for $P_{\tau(C)}$ is partially constructed.
2. In the generation phase, a first-order formula for C is created and grounded, and ASP rules are generated, including weight constraints when necessary.
3. In the execution phase an ASP system attempts to produce a stable model for $P_{\tau(C)}$. C is consistent if and only if such a model exists.

CDFTP consists of about 1400 lines of XSB code to perform the first two steps above, where XSB's tabling is used for grounding. Code can be generated for SModels for $\mathcal{ALCQI}$ expressions and for DLV for $\mathcal{ALCI}$ expressions. There is a *file-level* interfaces for DLV and SModels in which the generator writes out ASP rules to a file, that is then read by the ASP system. For SModels, a much faster C-level interface is possible that uses XSB's XASP interface and the SModels C API.

Algorithmic Optimizations CDFTP tries to generate as efficient and as small an ASP program as possible by reducing the nondeterminism of the ASP program, while leaving it to the ASP system to perform satisfiability checks. Some optimizations performed are

Partial evaluation of deterministic facts. Atoms in $ground(C)$ that occur positively and outside of the scope of a disjunction are necessary for the satisfaction of $ground(C)$. Such atoms need not be made non-deterministic, and can be removed from the bodies of any rules that include them. This optimization is implicit in Magic Asp (Section 3.2).

Goal-Oriented generation of structure elements. If a class expression contains a disjunction $D_1 \vee D_2$, the Magic Asp optimization ensures that structure elements needed for D_1 and for D_2 are generated in turn and with a minimum amount of non-deterministic search.

Analysis of Qualified Number Restrictions. As shown in Theorem 3 $\mathcal{ALCQ}$ expressions may not have a model whose structure graph is a tree. Accordingly, in Section 3.3 grounding of variables in a subsentence S makes use both of a set S_{new} of new constants generated for S and a set S_{global} of constants generated for sentences outside of S. Clearly, having to check both S_{new} and S_{global} is inefficient. CDFTP analyzes, which sentences generate atoms that may lie within the scope of an $atMost$ operator; these sentences are grounded with S_{new} and S_{global}, others are grounded with S_{new} alone.

Some Performance Results CDFTP was tested against version 3.3 of the DLP Description Logic System [1], using DLP's default compilation option. DLP is one of the fastest description logic theorem provers and is based on a tableaux-style proof technique and implemented in ML. According to its documentation, version 3.3 handles only $\mathcal{ALCN}$ expressions, so that number restrictions cannot be qualified. Benchmarking theorem provers is notoriously difficult to do well. Since most theorem provers address problems that are NP-complete or NP-hard, they make use of various heuristics for non-deterministic search. As a result, various problems may show those heuristics to advantage or disadvantage. Our benchmark results are not intended to indicate which theorem prover is better for which purposes; but rather is intended to provide a flavor of the performance of CDFTP.

Four benchmarks were chosen for this analysis. The first, $Chain_{nonsat}$, denotes a non-satisfiable formula $(exists(r, (exists(r, (\ldots (exists(r, (c \sqcap \neg c)))$ $\ldots)))))$ whose length (number of r relations traversed) was 32000. The second, $Chain_{sat}$ denotes a satisfiable formula $(exists(r, (exists(r, (\ldots (exists(r, \top))$

$\ldots)))))$ also of length 32000. $Tree_{nonsat}$ denotes non-satisfiable disjunction of the form $(exists(r, ((c \sqcap \neg c) \sqcup exists(r, (c \sqcap \neg c) \sqcup \ldots (exists(r, (c1 \sqcap \neg c1))) \ldots))))$ of length 3200. The final benchmark, $Tree_{sat}$ denotes a satisfiable disjunction $(exists(r, (c \sqcup exists(r, c \sqcup \ldots (exists(r, c)) \ldots))))$ of length 400. Results are shown in in Table 1.

Table 1. Results (in seconds) of the CDF Theorem Prover and DLP

	Analysis	Generation	Commit	SModels	CDF Total	DLP
$Chain_{nonsat}$	1.1	1.6	0.2	0.2	3.1	1.7
$Chain_{sat}$	1.1	1.8	0.2	0.2	3.3	6.4
$Tree_{nonsat}$	0.2	1.0	0.2	0.2	1.6	0.5
$Tree_{sat}$	0.0	0.1	0.0	1.8/(0.0)	1.95/(0.1)	0.1

The first four columns give indications of the time for CDFTP to perform the analysis phase, the generation phase, to transfer the generated ASP rules from XSBs interned format (asserted tries) to SModels interned format, and for SModels to either produce the first stable model or to indicate that no stable model existed. Despite the simplicity of these benchmark programs, a number of observations may be made. First, both theorem provers are robust enough to handle these benchmarks reasonably efficiently. On the chains, the time to analyze and generate rules for the large class expressions is larger than the time to check consistency for the ASP programs themselves. This is not unexpected, as the resulting programs have no inherent non-determinism. On the other hand, in analysis phase of CDFTP includes overheads to analyze qualified number restrictions and inverse relations that are not needed in any of these benchmarks. Surprisingly, the time for DLP on $Chain_{sat}$ is rather slow using the default configuration, possibly due to poor indexing within the tableaux it constructs. However, we note that DLP contains a number of other configuration options that might speed evaluation in this, or other examples.

For the trees, the analysis time is much smaller, though the generation time is somewhat more significant for $Tree_{nonsat}$ as numerous Magic ASP rules need to be generated (the tree benchmarks both take several minutes if Magic ASP is not used). For the first three benchmarks, though, the times to commit an ASP program to SModels and for SModels execution is very low; but for the fourth the default setting for SModels proved quite inefficient. To understand this, consider that SModels uses a default heuristic for pruning the search space called a *lookahead* heuristic. SModels tries to construct a stable model by iteratively choosing truth values for atoms, and adding these truth values to a partially constructed model. In order to choose which atom to add at each iteration it tries to pick an atom that will determine the truth value of a large number of other atoms. An alternative heuristic, used by DLP v. 3.3 and other tableaux theorem provers, is to use some sort of dependency-driven back-jumping. It is possible to invoke SModels to use back-jumping and not to use lookahead: when this is done the times, given in parentheses in Table 1 are comparable with DLP.

5 Discussion

ASP has advantages for implementing deduction in ontologies. It can be tightly coupled to Prolog programs, both at the level of semantics and of implementation, making it useful for ontology management systems like Cold Dead Fish that are implemented using Prolog. The clear logical semantics of ASP means that correctness proofs of a theorem prover reduces to the proof of correctness of a series of transformations, from class expressions, to sentences in first-order logic, to sentences in propositional logic and on to the ASP programs themselves. Furthermore, many properties useful to theorem provers can be formulated at the level of first-order sentences and proved using elementary results from model theory. Consequently, correctness proofs become much shorter than they would be otherwise. The resulting CDFTP implementation may prove useful when extending deduction to include relational properties (such as transitivity), default reasoning, and so on. While further optimizations are possible for CDFPT, its times in Section 4 indicated that the approach is scalable and certainly robust.

References

1. Patel-Schneider, P.: DLP. In: Proc. of the Intl Workshop on Description Logics. (1999)
2. Donini, F., Lenzerini, M., Nardi, D., Nutt, W.: The complexity of concept languages. Information and Computation (1996)
3. Schmidt-Strauss, M., Smolka, G.: Attributive concept descriptions with complements. Artificial Intelligence **48** (1990) 1–26
4. Hollunder, B., Nutt, W.: Subsumption algorithms for concept description languages. (1990)
5. Horrocks, I.: The FaCT system. In: International Conference Tableaux 98. Volume 1397 of LNCS. (1998) 307–312
6. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (2002)
7. Borgida, A.: On the relative expressiveness of description logics and predicate logics. **82** (1996) 353
8. Niemelä, I., Simons, P.: Smodels: An implementation of the stable and well-founded semantics for normal LP. Volume 1265 of LNAI. (1997) 420–429
9. Leone, N., Pfeifer, G., Faber, W.: The DLV Project. (2003) `http://www.dbai.tuwein.at/proj/dlv`.
10. Calvenese, D., Giacomo, G.D., Lenzerini, M., Nardi, D.: Reasoning in expressive description logics. Volume 2. (2002) 1582–1634
11. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234
12. Swift, T., Warren, D.S.: Cold Dead Fish: A System for Managing Ontologies. (2003) Available via `http://xsb.sourceforge.net`.

Strong Equivalence for Causal Theories

Hudson Turner*

Computer Science Department
University of Minnesota, Duluth
Duluth, MN 55812 USA
hudson@d.umn.edu

Abstract. Strong equivalence is an important property for nonmonotonic formalisms, allowing safe local changes to a nonmonotonic theory. This paper considers strong equivalence for nonmonotonic causal theories of the kind introduced by McCain and Turner. Causal theories T_1 and T_2 are strongly equivalent if, for every causal theory T, $T_1 \cup T$ and $T_2 \cup T$ are equivalent (that is, have the same causal models). The paper introduces a convenient characterization of this property in terms of so-called SE-models, much as was done previously for answer set programs and default theories. A similar result is provided for the nonmonotonic modal logic UCL. The paper also introduces a reduction from the problem of deciding strong equivalence of two causal theories to the problem of deciding equivalence of two sets of propositional formulas.

1 Introduction

Strong equivalence is an important property for nonmonotonic formalisms; it allows one to safely make local changes to a nonmonotonic theory (without having to consider the whole theory). The generic property can be defined thus: theories P and Q (in some nonmonotonic formalism) are *strongly equivalent* if, for every theory R, theories $P \cup R$ and $Q \cup R$ are equivalent. Such P can be safely replaced with Q, with no need to consider the context in which P occurs.

Lifschitz, Pearce and Valverde [13] introduced the notion of strong equivalence and used Heyting's logic of here-and-there to characterize strong equivalence for logic programs with nested expressions [14] (and, more generally, for equilibrium logic [19]). A closely related characterization in terms of so-called SE-models was introduced in [24]. In that paper, a variant of SE-models was also used to characterize strong equivalence for default logic [21,7]. In [25], SE-models were used to characterize strong equivalence for the weight constraint programs of Niemelä and Simons [18]. The current paper introduces another variant of SE-models to characterize strong equivalence for nonmonotonic causal theories.

Nonmonotonic causal theories were introduced by McCain and Turner in [17]. The current paper considers the extension of causal theories, introduced in [8], in which the atomic parts of formulas are equalities of a kind that may be found in constraint satisfaction problems. This is convenient when formulas are used to

* Partially supported by NSF Career Grant #0091773.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 289–301, 2004.

talk about states of a system. For instance, to describe the location of a person in an apartment, we can use equalities like

$$Loc = Kitchen, Loc = LivingRoom, Loc = Bathroom, Loc = Bedroom.$$

The direct effect of walking to the kitchen can then be described simply by saying that there is a cause for the first of these atomic formulas to be true (from which it will follow that there is a cause for the others to be false). By comparison, in a strictly Boolean language, the relationship between these atomic formulas must be expressed by additional formulas; in a case like this, one must express, so to speak, not only the logical relationship between them (exactly one is true at each moment) but also the causal relationship.

An implementation of causal theories—the Causal Calculator (CCALC)[1]—has been applied to several challenge problems in the theory of commonsense knowledge [12,11,1,5], to wire-routing problems [6], and to formalization of multi-agent computational systems [2,3]. Lee and Lifschitz [9] have shown how to use causal theories to represent "additive" fluents—fluents with numerical values that may be affected by concurrent actions whose interacting effects must be suitably (often additively) combined. In working out these various applications, our understanding of the causal theories formalism has begun to mature.

The mathematical results of the current paper contribute to this maturation process. In addition to providing a mathematically simple characterization of strong equivalence for causal theories, the paper describes an easy reduction of the problem of deciding strong equivalence of two causal theories to the problem of deciding equivalence of two comparably-sized sets formulas of propositional logic. This makes it straightforward to check strong equivalence of finite causal theories automatically, via standard satisfiability solvers.

The paper also includes a characterization of strong equivalence for the non-monotonic modal logic UCL [23], which can be understood as an extension of the causal theories formalism. One of the initial motivations for the introduction of UCL was to provide a more adequate semantic foundation for causal theories. UCL is obtained by imposing a simple fixpoint condition on standard S5 modal logic. Thus, standard S5 provides a basis in UCL for the kinds of replacement properties guaranteed by strong equivalence. That is, in UCL one can safely replace any subtheory with an S5-equivalent, without changing the set of causal models. What we will see is that the SE-models for UCL are actually a slight specialization of the S5-models. (So S5-equivalence implies strong equivalence, but the converse does not hold.)

We proceed as follows. Section 2 reviews the class of propositional formulas from which causal theories and UCL will be built. Section 3 reviews the syntax and semantics of causal theories. Section 4 introduces and discusses the SE-model characterization of strong equivalence for causal theories. Section 5 presents the reduction from strong equivalence of causal theories to equivalence of sets of propositional formulas. Section 6 introduces the syntax and semantics of UCL, and shows that causal theories are subsumed by UCL. Section 7 presents the

[1] `http://www.cs.utexas.edu/users/tag/cc/`

SE-model characterization of strong equivalence for UCL. Section 8 consists of concluding remarks.

2 Formulas

Following [8], the class of formulas defined here is similar to the class of propositional formulas, but a little bit more general: we will allow atomic parts of a formula to be equalities of the kind found in constraint satisfaction problems.

A (multi-valued propositional) *signature* is a set σ of symbols called *constants*, along with a nonempty finite set $Dom(c)$ of symbols, disjoint from σ, assigned to each constant c. We call $Dom(c)$ the *domain* of c. An *atom* of a signature σ is an expression of the form

$$c{=}v$$

("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *formula* over σ is a propositional combination of atoms.

An *interpretation* of σ is a function that maps every element of σ to an element of its domain. An interpretation I *satisfies* an atom $c{=}v$ if $I(c) = v$. The satisfaction relation is extended from atoms to arbitrary formulas according to the usual truth tables for the propositional connectives.

As usual, the symbol $\models$ denotes the satisfaction relation. An interpretation *satisfies* a set of formulas if it satisfies each formula in the set. An interpretation that satisfies a formula, or a set of formulas, is said to be one of its *models*. Formulas or sets of formulas are *equivalent* if they have the same models.

A *Boolean* constant is one whose domain is the set $\{\mathbf{f}, \mathbf{t}\}$ of truth values. A *Boolean* signature is one whose constants are Boolean. If c is a Boolean constant, we will sometimes write c as shorthand for the atom $c{=}\mathbf{t}$. Under this convention, when the syntax and semantics defined above are restricted to Boolean signatures and to formulas that do not contain $\mathbf{f}$, they turn into the usual syntax and semantics of classical propositional formulas.

3 Causal Theories

3.1 Syntax

Begin with a multi-valued propositional signature σ. By a (causal) *rule* we mean an expression of the form

$$F \Leftarrow G$$

("F is caused if G is true") where F and G are formulas over σ, called the *head* and the *body* of the rule. A *causal theory* is a set of causal rules.

3.2 Semantics

For any causal theory T and interpretation I of its signature, let

$$T^I = \{\, F : F \Leftarrow G \in T,\, I \models G \,\} \ .$$

We say that I is a *causal model* of T if I is the unique model of T^I.

3.3 Examples, Remarks, an Auxiliary Definition, and Two Facts

The following example makes use of the convention that for a Boolean constant c, we can write c as shorthand for the atom $c{=}\mathbf{t}$.

Example 1. Let T be the causal theory over the Boolean signature $\{p, q\}$ whose rules are

$$\begin{aligned} p &\Leftarrow q \ , \\ q &\Leftarrow q \ , \\ \neg q &\Leftarrow \neg q \ . \end{aligned}$$

Of the four interpretations of this signature, only the interpretation I_1 that maps both p and q to $\mathbf{t}$ is a causal model. Indeed, $T^{I_1} = \{p, q\}$, whose unique model is I_1. Let I_2 be such that $I_2(p) = \mathbf{f}$ and $I_2(q) = \mathbf{t}$. Then $T^{I_2} = \{p, q\}$, which is not satisfied by I_2, which is therefore not a causal model of T. Finally, if I maps q to $\mathbf{f}$, then $T^I = \{\neg q\}$, which has two models. Consequently, such an I cannot be a causal model of T.

For technical convenience, we extend the definition of satisfaction to causal rules and thus to causal theories; to this end, $I \models F \Leftarrow G$ if $I \models G \supset F$.

In defining satisfaction of causal theories thus, we certainly do not mean to suggest that the causal connective $\Leftarrow$ can be identified with the material conditional $\supset$. To the contrary, notice for instance that the latter two rules in Example 1 are satisfied by every interpretation of the signature, and yet each describes a distinct condition under which a distinct state of affairs has a cause.

The following observation is easily verified.

Fact 1 *For any causal theory T and interpretation I, $I \models T$ iff $I \models T^I$.*

The following easy corollary can also be useful.

Fact 2 *For any causal theory T and interpretation I, if I is a causal model of T, then $I \models T$.*

Next we consider an example that makes use of non-Boolean constants.

Example 2. Take $\sigma = \{a, b\}$ with $Dom(a) = \{0, 1\}$ and $Dom(b) = \{0, 1, 2\}$. Let

$$T = \left\{ \begin{array}{c} a{=}0 \not\equiv b{=}0 \Leftarrow \top \ , \\ a{=}1 \Leftarrow a{=}1 \ , \\ \neg b{=}1 \Leftarrow b{=}2 \ , \\ \bot \Leftarrow a{=}0 \wedge b{=}1 \end{array} \right\} \ .$$

For each $j \in \{0, 1\}$ and $k \in \{0, 1, 2\}$, let I_{jk} be the interpretation of σ such that $I_{jk}(a) = j$ and $I_{jk}(b) = k$. Notice that $I_{00} \not\models a{=}0 \not\equiv b{=}0 \Leftarrow \top$, so $I_{00} \not\models T$, and we can conclude by Fact 2 that I_{00} is not a causal model of T. For the same reason, I_{11} and I_{12} are not causal models of T. And since $I_{01} \not\models \bot \Leftarrow a{=}0 \wedge b{=}1$,

Fact 2 also shows that I_{01} is not a causal model of T. For the remaining interpretations, we have

$$T^{I_{02}} = \{\, a{=}0 \not\equiv b{=}0,\ \neg b{=}1 \,\}\ ,$$
$$T^{I_{10}} = \{\, a{=}0 \not\equiv b{=}0,\ a{=}1 \,\}\ .$$

$T^{I_{02}}$ has two models (namely, I_{02} and I_{10}), so I_{02} is not a causal model of T. The unique model of $T^{I_{10}}$ is I_{10}, so I_{10} is a causal model of T.

The definition of causal models reflects the following intuitions (with respect to those features of worlds that are described by the constants in the signature of causal theory T).

- For each interpretation I, T^I is a description of exactly what is caused in worlds like I.
- The "causally explained" worlds are those in which (i) everything that is caused obtains and, moreover, (ii) everything that obtains is caused.

Accordingly, I is a causal model of T iff $I \models T^I$ and, so to speak, everything about I has a cause according to T (that is, no interpretation different from I satisfies T^I). Requirement (i) is not surprising. Requirement (ii) is more interesting, and since it is the key to the mathematically simple fixpoint definition, we go so far as to name it "the principle of universal causation." In practice, it is often accomodated in part by writing rules of the form

$$F \Leftarrow F$$

which can be understood to stipulate that F has a cause whenever F is the case.

Similarly, in applications of causal theories to reasoning about action, the commonsense law of inertia is most typically expressed by rules of the form

$$t{+}1{:}c{=}v \Leftarrow t{+}1{:}c{=}v \wedge t{:}c{=}v$$

which can be understood to stipulate that if c has value v at time t and keeps that value at time $t+1$ then there is a cause for $c{=}v$ at time $t+1$. (Intuitively, the cause is inertia.) As explored elsewhere, such rules provide a robust solution to the frame problem in the context of causal theories.

For further discussion, see [17,23,8]. Additional examples can also be found in the application papers cited in the introduction.

4 Strong Equivalence for Causal Theories

Causal theories T_1 and T_2 are *equivalent* if they have the same causal models.

Causal theories T_1 and T_2 are *strongly equivalent* if, for every causal theory T, $T_1 \cup T$ and $T_2 \cup T$ are equivalent.

An *SE-structure* for causal theories over signature σ is simply a pair of interpretations of σ. An SE-structure (I, J) is an *SE-model* of causal theory T if $J \models T$ and $I \models T^J$.

Theorem 1. *Causal theories are strongly equivalent iff they have the same SE-models.*

Hence, strongly equivalent causal theories not only have the same satisfying interpretations but also agree on what is caused with respect to each of them.

It is clear that strong equivalence is maintained under substitution of equivalent formulas in heads and bodies of rules. What is more interesting is determining when one set of rules can be safely replaced with another.

The following result appears as Proposition 4 in [8].

Fact 3 (*i*) *Replacing a rule*

$$F \wedge G \Leftarrow H$$

in a causal theory by the rules

$$F \Leftarrow H,\ G \Leftarrow H$$

does not affect its causal models. (*ii*) *Replacing a rule*

$$F \Leftarrow G \vee H$$

in a causal theory by the rules

$$F \Leftarrow G,\ F \Leftarrow H$$

does not affect its causal models.

These are essentially claims about strong equivalence, and they are easily checked using Theorem 1. For instance, to check part (*i*), consider any SE-structure (I, J). Notice first that $J \models F \wedge G \Leftarrow H$ iff $J \models F \Leftarrow H$ and $J \models G \Leftarrow H$. Then notice that $I \models F \wedge G$ iff $I \models F$ and $I \models G$.

The following result shows that replacement of a formula F by a formula G is safe in the presence of the rule $F \equiv G \Leftarrow \top$.

Proposition 1. *Let T_1 and T_2 be causal theories, with T_2 obtained from T_1 by replacing any number of occurrences of formula F by formula G. Let*

$$E = \{\, F \equiv G \Leftarrow \top \,\} \ .$$

Then $T_1 \cup E$ and $T_2 \cup E$ are strongly equivalent.

This proposition follows easily from Theorem 1, given that the replacement property holds for formulas.[2]

The choice of the rule $F \equiv G \Leftarrow \top$ in the statement of Proposition 1 is optimal in the sense that adding it to T_1 "eliminates" all and only those SE-models (I, J) of T_1 for which either $I \not\models F \equiv G$ or $J \not\models F \equiv G$.

Before presenting the proof of Theorem 1, we consider one more example of its use.

To this end, let us agree to call a causal theory *conjunctive* if the head of each of its rules is a conjunction (possibly empty) of atoms.

[2] That is, if F_1 and F_2 are formulas, with F_2 obtained from F_1 by replacing any number of occurrences of formula F by formula G, and $I \models F \equiv G$, then $I \models F_1 \equiv F_2$.

Proposition 2. *Let $c_1 = v_1$ and $c_2 = v_2$ be distinct atoms that can be jointly falsified. (Constants c_1 and c_2 need not be distinct.) The causal theory*

$$\{\, c_1 = v_1 \vee c_2 = v_2 \Leftarrow \top \,\}$$

is not strongly equivalent to any conjunctive causal theory.[3]

Proof. Let T be the causal theory above, and suppose that T_c is a strongly equivalent conjunctive causal theory. (We will derive a contradiction.) Take interpretations I and J such that I satisfies only the first of the atoms in T and J satisfies only the second. (This is possible since the atoms are distinct and each can be falsified.) Both (I, J) and (J, J) are SE-models of T. By Theorem 1, (I, J) and (J, J) are SE-models of T_c. Hence, $I \models T_c^J$ and $J \models T_c^J$. Since each element of T_c^J is a conjunction of atoms, we can conclude that c_1 does not appear in T_c^J; indeed, any atom involving constant c_1 would be falsified by either I or J, since they were chosen to disagree on c_1. Similarly, c_2 cannot appear in T_c^J. And since there are no more constants in the signature, and T_c^J is a satisfiable set of conjunctions of atoms, we conclude that $T_c^J = \emptyset$. Let I_0 be an interpretation that falsifies both $c_1 = v_1$ and $c_2 = v_2$. (By assumption they are jointly falsifiable.) Then (I_0, J) is an SE-model of T_c but not of T, which by Theorem 1 yields the contradiction we seek. □

Now let us consider the proof of Theorem 1, which is quite similar to the proofs of the corresponding strong equivalence theorems from [24,25] for logic programs with nested expressions, default theories, and weight constraint programs.

The following lemma is easily verified.

Lemma 1. *Causal theories with the same SE-models have the same causal models.*

Proof (of Theorem 1). Assume that causal theories T_1 and T_2 have the same SE-models. We need to show that they are strongly equivalent. So consider an arbitrary causal theory T. We need to show that $T_1 \cup T$ and $T_2 \cup T$ have the same causal models. Since T_1 and T_2 have the same SE-models, so do $T_1 \cup T$ and $T_2 \cup T$, and it follows by Lemma 1 that $T_1 \cup T$ and $T_2 \cup T$ also have the same causal models.

Now assume that T_1 and T_2 have different SE-models. Without loss of generality, assume that (I, J) is an SE-model of T_1 and not of T_2. We need to show that T_1 and T_2 are not strongly equivalent. Consider two cases.

Case 1: $J \not\models T_2$. Take

$$T = \{\, c = v \Leftarrow \top \,:\, J(c) = v \,\}\ .$$

Because (I, J) is an SE-model of T_1, $J \models T_1$, and by Fact 1, $J \models T_1^J$. Meanwhile, it is clear that J is the unique model of T^J. Consequently, J is the unique model

[3] A similar result for logic programs appears in [25].

of $T_1^J \cup T^J = (T_1 \cup T)^J$. That is, J is a causal model of $T_1 \cup T$. On the other hand, since $J \not\models T_2$, $J \not\models T_2 \cup T$, and it follows by Fact 2 that J is not a causal model of $T_2 \cup T$. Hence, in this case, T_1 and T_2 are not strongly equivalent.

Case 2: $J \models T_2$. It follows that $I \not\models T_2^J$, since (I, J) is not an SE-model of T_2. On the other hand, it follows by Fact 1 that $J \models T_2^J$. We may conclude that $I \neq J$. Take

$$T = \{\, \neg c_1{=}v_1 \supset c_2{=}v_2 \Leftarrow \top \,:\, I(c_1) = v_1,\, J(c_2) = v_2 \,\}\ .$$

Notice that the only models of T^J are I and J. Since $I \not\models T_2^J$ while $J \models T_2^J$, we may conclude that J is the unique model of $T_2^J \cup T^J = (T_2 \cup T)^J$. That is, J is a causal model of $T_2 \cup T$. On the other hand, since (I, J) is an SE model of T_1, $I \models T_1^J$. We may conclude that $I \models T_1^J \cup T^J = (T_1 \cup T)^J$, and since $I \neq J$, it follows that J is not a causal model of $T_1 \cup T$. Hence, in this case too, T_1 and T_2 are not strongly equivalent. □

5 Strong Equivalence of Causal Theories as Equivalence of Sets of Propositional Formulas

As we will show, strong equivalence of two causal theories can be determined by checking the equivalence of two corresponding, comparably-sized sets of propositional formulas. Consequently, if the causal theories are finite, we can decide strong equivalence by deciding the unsatisfiability of a corresponding propositional formula. Of course this can in turn be decided by a standard satisfiability solver, once we reduce the multi-valued propositional formula to a Boolean propositional formula (which is straightforward).

The idea of the encoding is first to make a "copy" of the signature, since we are interested in SE-structures, which are pairs of interpretations.

For any signature σ, let signature $\sigma' = \{c' : c \in \sigma\}$, where c' is unique for each $c \in \sigma$, and σ' is disjoint from σ, with $Dom(c') = Dom(c)$ for each $c \in \sigma$. For any interpretation I of σ, let I' be the interpretation of σ' such that $I'(c') = I(c)$ for all $c \in \sigma$.

Thus, the SE-structures for signature σ are in one-to-one correspondence with the interpretations of $\sigma' \cup \sigma$, each of which can be represented in the form $I' \cup J$ for some SE-structure (I, J).

Of course there is no problem encoding the first SE-model condition, $J \models T$, and all that is needed for the second condition, $I \models T^J$, is to make a "copy" of the head of each rule as in the following.

For any causal theory T over signature σ,

$$se(T) = \{\, G \supset F \wedge F' \,:\, F \Leftarrow G \in T \,\}$$

where F' is the formula obtained from F by replacing each occurrence of each constant $c \in \sigma$ with the corresponding constant $c' \in \sigma'$. So $se(T)$ is a set of formulas over $\sigma' \cup \sigma$.

Lemma 2. *For any causal theory T, an SE-structure (I, J) is an SE-model of T iff $I' \cup J \models se(T)$.*

Proof. Observe that the following two conditions are equivalent.

- $J \models F \Leftarrow G$, and
  if $J \models G$ then $I \models F$.
- $I' \cup J \models G \supset F \wedge F'$.

Consequently, $J \models T$ and $I \models T^J$ iff $I' \cup J \models se(T)$. □

Theorem 2. *Causal theories T_1 and T_2 are strongly equivalent iff $se(T_1)$ and $se(T_2)$ are equivalent.*

Proof. Follows easily from Lemma 2, given that the mapping $(I, J) \mapsto I' \cup J$ is a bijection from the set of SE-structures for σ—the signature of T_1 and T_2—to the set of interpretations of $\sigma' \cup \sigma$—the signature of $se(T_1)$ and $se(T_2)$. □

Notice that if T is finite, so is $se(T)$. Abusing notation slightly, let us understand $se(T)$ to stand for the conjunction of its elements in the case that T is finite. Then we have the following corollary to Theorem 2.

Corollary 1. *Finite causal theories T_1 and T_2 are strongly equivalent iff the formula $se(T_1) \not\equiv se(T_2)$ is unsatisfiable.*

All that remains in order to automate the decision process is to reduce the formula $se(T_1) \not\equiv se(T_2)$ to a corresponding Boolean propositional formula, which is straightforward. (See Appendix A in [8], for instance.)

The encoding above closely resembles previous encodings [20,15] for reducing strong equivalence of logic programs to equivalence in classical propositional logic. It is also related to the reduction in [25] of strong equivalence of two weight constraint programs to inconsistency of a single corresponding weight constraint program.

6 UCL

UCL is a modal nonmonotonic logic obtained from standard S5 modal logic by imposing a simple fixpoint condition that reflects the "principle of universal causation" (discussed in Sect. 3 in connection with causal theories). In [23], UCL was defined not only in the (Boolean) propositional case, but also for nonpropositional languages that include first and second-order quantifiers. In the current paper, we consider a different extension of (Boolean) propositional UCL, built from the multi-valued propositional formulas defined in Sect. 2.

The fundamental distinction in UCL—between propositions that have a cause and propositions that (merely) obtain—is expressed by means of the modal operator C, read as "caused." For example, one can write

$$G \supset \mathsf{C}F \tag{1}$$

to say that F is caused whenever G obtains. If we assume that F and G are formulas of the kind defined in Sect. 2, then formula (1) corresponds to the causal rule $F \Leftarrow G$. This claim is made precise in Theorem 3 below.

6.1 Syntax

UCL formulas are obtained by extending the (implicit) recursive definition of formulas from Sect. 2 with an additional case for the modal operator C, in the usual way for modal logic:

– If F is a UCL formula, then so is $\mathsf{C}F$.

A *UCL theory* is a set of UCL formulas.

6.2 Semantics

An *S5-structure* is a pair (I, S) such that I is an interpretation and S is a set of interpretations (all of the same signature) to which I belongs. *Satisfaction* of a UCL formula by an S5-structure is defined by the standard recursions over the propositional connectives, plus the following two conditions:

– if p is an atom, $(I, S) \models p$ iff $I \models p$,
– $(I, S) \models \mathsf{C}F$ iff, for all $J \in S$, $(J, S) \models F$.

We call the S5-structures that satisfy a UCL formula its *S5-models*. UCL formulas are *S5-equivalent* if they have the same S5-models. We extend these notions to UCL theories (that is, sets of UCL formulas) in the usual way.

For a UCL theory T, if $(I, S) \models T$, we say that (I, S) is an *I-model* of T, thus emphasizing the distinguished interpretation I.

We say that I is *causal model* of T if $(I, \{I\})$ is the unique I-model of T.

6.3 UCL Subsumes Causal Theories

Theorem 3. *For any causal theory T, the causal models of T are precisely the causal models of the corresponding UCL theory*

$$\{\, G \supset \mathsf{C}F \,:\, F \Leftarrow G \in T \,\}\ .$$

The proof presented here is essentially the same as the proof given in [23] for the (Boolean) propositional case, despite the fact that the current paper uses a different, more general definition of a propositional formula.

We begin the proof with a lemma that follows easily from the definitions.

Lemma 3. *For every causal rule $F \Leftarrow G$ and S5-structure (I, S), the following two conditions are equivalent.*

– $(I, S) \models G \supset \mathsf{C}F$.
– *If $I \models G$, then, for all $J \in S$, $J \models F$.*

Proof (of Theorem 3). Let T' be the UCL theory corresponding to causal theory T.

Assume that I is the unique model of T^I. By Lemma 3, $(I, \{I\}) \models T'$. Let S be a superset of $\{I\}$ such that $(I, S) \models T'$. By Lemma 3, for all $J \in S$, $J \models T^I$. It follows that $S = \{I\}$, so $(I, \{I\})$ is the unique I-model of T'.

Assume that $(I, \{I\})$ is the unique I-model of T'. By Lemma 3, $I \models T^I$. Assume that $J \models T^I$. By Lemma 3, $(I, \{I, J\}) \models T$. It follows that $I = J$, so I is the unique model of T^I. □

A similar result in [23] shows that UCL with first and second-order quantification subsumes the nonpropositional causal theories of [10], which in turn subsume the (Boolean) propositional causal theories of [17].

As mentioned previously, one of the original motivations for UCL was to provide a more adequate semantic foundation for causal theories. Another was to unify two closely related approaches to action representation, one based on causal theories and the other based essentially on default logic [16,22]. See [23] for more on UCL, including its close relationship to disjunctive default logic [7] (an extension of Reiter's default logic).

7 Strong Equivalence for UCL

It is clear that S5-equivalence implies strong equivalence for UCL theories, but the converse does not hold. For example, let $T = \{\neg \mathsf{C}p, \neg \mathsf{C}\neg p\}$. No superset of T has a causal model, so all supersets of T are strongly equivalent, yet T is not S5-equivalent to $T \cup \{p\}$, for instance.

To capture strong equivalence, we slightly strengthen the notion of S5-model.

An S5-model (I, S) of UCL theory T is an *SE-model* of T if $(I, \{I\}) \models T$.

That is, (I, S) is an SE-model iff both $(I, \{I\})$ and (I, S) are S5-models. Notice, for instance, that the example UCL theory T above has no SE-models, which is consistent with the observation that no superset of T has a causal model.

Theorem 4. *UCL theories are strongly equivalent iff they have the same SE-models.*

The proof of this theorem is similar to the proofs of SE-model characterizations of strong equivalence for other nonmonotonic formalisms. In particular, it is very much like the earlier proof of Theorem 1 (for causal theories), and so is given a sketchier presentation.

We begin with the usual easily verified lemma.

Lemma 4. *UCL theories with the same SE-models have the same causal models.*

Proof (of Theorem 4). The right-to-left direction is just as in the proof of Theorem 1, except that now we are interested in UCL theories (instead of causal theories) and we use Lemma 4 (instead of Lemma 1).

For the other direction, assume without loss of generality that (I, S) is an SE-model of T_1 but not of T_2.

Case 1: $(I, \{I\}) \not\models T_2$. Take

$$T = \{\, \mathsf{C}c{=}v \,:\, I(c) = v \,\}$$

and the rest is easy. (See the corresponding case in the proof of Theorem 1).

Case 2: $(I, \{I\}) \models T_2$. So $(I, S) \not\models T_2$, and thus $S \neq \{I\}$. Take

$$T = \{\, F \supset \mathsf{C}c{=}v \,:\, F \in T_2,\, (I, S) \not\models F,\, I(c) = v \,\} \ .$$

Notice first that $(I, S) \models T_1 \cup T$, and since $S \neq \{I\}$, I is not a causal model of $T_1 \cup T$. Notice next that $(I, \{I\}) \models T_2 \cup T$. Assume that $(I, S') \models T_2 \cup T$. Since $(I, S) \not\models T_2$ and $(I, S') \models T_2$, there is an $F \in T_2$ such that $(I, S) \not\models F$ and $(I, S') \models F$. And since $(I, S') \models T$, we may conclude that $S' = \{I\}$. So I is a causal model of $T_2 \cup T$. □

8 Concluding Remarks

This paper introduces mathematically simple characterizations of strong equivalence for causal theories and for the nonmonotonic modal logic UCL. (It also shows that even when multi-valued propositional atoms are incorporated, UCL subsumes causal theories, as expected.)

One of the original motivations for UCL was to provide a more adequate semantic foundation for causal theories, thus securing replacement properties like those guaranteed by strong equivalence. The results of this paper imply that, from this point of view, (propositional) UCL was quite successful. Indeed, from Lemma 3 it follows that causal theories have the same SE-models iff the corresponding UCL theories have the same S5-models. (To see this, let T' be the UCL theory corresponding to causal theory T, and use Lemma 3 to verify that $(J, S) \models T'$ iff, for all $I \in S$, (I, J) is an SE-model of T.) Consequently, S5 modal logic, on which the fixpoint semantics of UCL is based, is perfectly adequate for characterizing strong equivalence of causal theories. [4]

At any rate, it is nice to have the simpler SE-models used to characterize strong equivalence of causal theories in Theorem 1. Moreover, they lead naturally to the result of Theorem 2, which reduces strong equivalence of causal theories to a question of equivalence of comparably-sized sets of propositional formulas.

References

[1] Varol Akman, Selim Erdoğan, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Representing the Zoo World and the Traffic World in the language of the Causal Calculator. *Artificial Intelligence*, 2003. To appear.

[2] A. Artikis, M. Sergot, and J. Pitt. An executable specification of an argumentation protocol. In *Proc. of Artificial Intelligence and Law (ICAIL)*, pages 1–11, 2003.

[3] A. Artikis, M. Sergot, and J. Pitt. Specifying electronic societies with the Causal Calculator. In *Proc. of Workshop on Agent-Oriented Software III (AOSE)*. Springer LNCS 2585, pages 1–15, 2003.

[4] Alex Bochman. A logic for causal reasoning. In *Proc. IJCAI'03*, pages 141–146, 2003.

[4] Bochman [4] independently establishes this fact (along with many others), expressed rather differently (in terms of smallest sets of rules closed under an inference system).

[5] J. Campbell and V. Lifschitz. Reinforcing a claim in commonsense reasoning. In *Logical Formalizations of Commonsense Reasoning: Papers from 2003 AAAI Spring Symposium*, pages 51–56, 2003.
[6] Esra Erdem, Vladimir Lifschitz, and Martin Wong. Wire routing and satisfiability planning. In *Proc. CL-2000*, pages 822–836, 2000.
[7] Michael Gelfond, Vladimir Lifschitz, Halina Przymusińska, and Miroslaw Truszczyński. Disjunctive defaults. In *Proc. KR'91*, pages 230–237, 1991.
[8] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 2003. To appear.
[9] Joohyung Lee and Vladimir Lifschitz. Describing additive fluents in action language C+. In *Proc. IJCAI'03*, pages 1079–1084, 2003.
[10] Vladimir Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.
[11] Vladimir Lifschitz. Missionaries and cannibals in the Causal Calculator. In *Proc. KR'00*, pages 85–96, 2000.
[12] Vladimir Lifschitz, Norman McCain, Emilio Remolina, and Armando Tacchella. Getting to the airport: the oldest planning problem in AI. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 147–165. Kluwer, 2000.
[13] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
[14] Vladimir Lifschitz, L.R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(2–3):369–390, 1999.
[15] Fangzhen Lin. Reducing strong equivalence of logic programs to entailment in classical logic. In *Proc. KR'02*, pages 170–176, 2002.
[16] Norman McCain and Hudson Turner. A causal theory of ramifications and qualifications. In *Proc. of IJCAI-95*, pages 1978–1984, 1995.
[17] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.
[18] Ilkka Niemelä and Patrik Simons. Extending the SMODELS system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.
[19] David Pearce A new logical characterization of stable models and answer sets. In *Non-Monotonic Extensions of Logic Programming (Lecture Notes in Artificial Intelligence 1216)*, pages 57–70, 1997.
[20] David Pearce, Hans Tompits, and Stefan Woltran. Encodings for equilibrium logic and logic programs with nested expressions. In *Proc. of the 10th Portuguese Conf. on AI (Lecture Notes in Artificial Intelligence 2258)*, pages 306–320, 2001.
[21] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.
[22] Hudson Turner. Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, 31(1–3):245–298, 1997.
[23] Hudson Turner. A logic of universal causation. *Artificial Intelligence*, 113:87–123, 1999.
[24] Hudson Turner. Strong equivalence for logic programs and default theories (made easy). In *Logic Programming and Nonmonotonic Reasoning: Proc. of Sixth Int'l Conf. (Lecture Notes in Artificial Intelligence 2173)*, pages 81–92, 2001.
[25] Hudson Turner. Strong equivalence made easy: Nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4&5):609–622, 2003.

Answer Set Programming with Clause Learning

Jeffrey Ward[1] and John S. Schlipf[2]

[1] The Ohio State University
jward@ececs.uc.edu
[2] University of Cincinnati, USA
john.schlipf@uc.edu

Abstract. A conflict clause represents a backtracking solver's analysis of why a conflict occurred. This analysis can be used to further prune the search space and to direct the search heuristic. The use of such clauses has been very important in improving the efficiency of satisfiability (SAT) solvers over the past few years, especially on structured problems coming from applications. We describe how we have adapted conflict clause techniques for use in the answer set solver Smodels. We experimentally compare the resulting program to the original Smodels program. We also compare to ASSAT and Cmodels, which take a different approach to adding clauses to constrain an answer set search.

1 Introduction

Recent years have seen the development of several stable model/answer set solvers. Smodels [Sim00,NSS00,Nie99] and DLV [EFLP00] are commonly used. (DLV implements more, namely disjunctive logic programming. However, it also serves as an effective stable model search engine.) These have demonstrated the feasibility of answer set programming as a paradigm for applications.

These programs have built upon the earlier propositional CNF satisfiability (SAT) solvers. But as the technology of the answer set programming systems has improved, the SAT solvers have gone on to implement new techniques, noticeably conflict clause usage (also known as "lemmas"), a variety of new search heuristics (which are frequently based on conflict clauses), and new highly efficient data structures. Key to many of these applications seems to be that some variables, and some combinations of variables, are far more important than others. More recent SAT solvers such as GRASP [MS99], SATO [Zheng97], rel_sat [BS97], Chaff [MMZZM01], BerkMin [GN02], and SIMO [GMT03], through creating and processing conflict clauses, often "learn" important information for the search.

The Cmodels–1 solver [Cmod] addresses this problem by piggy-backing an answer set solver onto a SAT solver. It handles a class of logic programs called *tight* [Fages94, BEL00], in which the stable models are just the models of Clark's completion of the program [Clark78] — and the completion is a classical logic problem. So Cmodels–1, after some preprocessing, passes a completion to a SAT solver, such as Chaff. Our concern in this paper is with solvers which are not limited to tight programs, so we do not discuss Cmodels–1 further here.[1]

[1] A different tool, by East and Truszczyński, takes a sort of middle ground between Cmodels–1 and ASSAT, but we shall not discuss it here either.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 302–313, 2004.

ASSAT [LZ02,ASSAT] drops the restriction to tight programs by an iterative process of calling a SAT solver, evaluating whether the model produced is stable, and, if not, adding further specifications ("loop formulas") to the problem to avoid this failure of stability. Recently, Cmodels–2 [BM03] adapts the technique of ASSAT, but extends its application to extended and disjunctive rules, and makes available alternative loop formula definitions.

Here we present an answer set programming tool (for non-disjunctive answer set programs), *Smodels*$_{cc}$ (Smodels with conflict clauses), that deals with the new technologies in SAT solvers in a different way. Instead of calling a fast SAT solver to perform the search, it incorporates some of the techniques of modern SAT solvers into a variant of Smodels. Like ASSAT, Smodels$_{cc}$ is intended to deal with non-tight, as well as tight, logic programs. It turns out that it is often much faster than the ASSAT approach on non-tight programs, since it incorporates the unfounded set calculation directly into the search engine, thus allowing tighter pruning of the search tree.

2 Background

Propositional CNF-SAT solvers. A basic *Davis-Putnam-Loveland-Logeman* $(DPLL)$ SAT solver [DLL62], given a set C of clauses, performs a backtracking search to find a model for C. We sketch it below as a recursive function, passed a set of literals representing a partial truth assignment. Initially, the partial assignment is empty. A *unit clause* is a not-yet-satisfied clause containing only one unassigned literal (which is thus forced); searching regularly for unit clauses and immediately inferring those literals, called *unit-propagation*, is almost universally done. In practice, literals forced by unit propagation are put into an "inference queue" until the data structures (not shown here) are updated; a contradiction is always revealed by inferring contradictory literals.

```
srch4ModlExtndng (partlAssgn)
   while there exists a unit clause c
      let unitLit be the remaining literal in c
      partlAssgn := partlAssgn union {unitLit}
      if any clause has been falsified, return (* backtrack *)
   if partlAssgn is total (contains each variable or its negation)
      output ‘‘SAT’’, output partlAssgn, and halt program
   else
      pick ltrl to guess next (i.e., branch on) - by a heuristic
      srch4ModlExtndng(partlAssgn union {ltrl})
      srch4ModlExtndng(partlAssgn union {not ltrl})
   if  partlAssgn is empty (* back at top level *) output ‘‘unSAT’’
```

Recent solvers add *conflict clause learning*. When Chaff infers a contradiction, it finds a *relatively* small $\{\lambda_1, \lambda_2, \ldots, \lambda_m\} \subseteq$ *partlAssgn* leading to that contradiction and, functionally, adds to C the *conflict clause* (a.k.a., *lemma*) $cc = (\neg\lambda_1 \vee \neg\lambda_2 \vee \cdots \vee \neg\lambda_m)$; cc is always a resolvent of clauses in C. It then backtracks (*backjumps*) to whatever level in the search tree unit propagation was executed after the *second to the last* of $\lambda_1, \lambda_2, \ldots, \lambda_m$ was added and simply restarts its search from this point. Since cc has been added to C, once all but one of the λ_i's are ever inferred again, the final $\neg\lambda_j$ will

be inferred, *before further search*, by unit propagation — converting the DPLL search tree to a DAG.[2] Some solvers also use the conflict clauses in their heuristics to choose literals to branch on (*i.e.*, guess next). An oversimplification is that Chaff branches on the unassigned literal occurring in the most conflict clauses.

Since so many conflict clauses are generated, systems must deal with storage overflow. Also, searching a huge store of conflict clauses for unit-clauses is very time-consuming. Chaff and BerkMin periodically completely restart their searches, keeping only those conflict clauses that, by some heuristic, seem likely to be useful. The cache of conflict clauses is then garbage collected and recompacted (improving data locality). Smodels$_{cc}$ also restarts, but it discards clauses continuously throughout the search and does no recompaction. Currently, Smodels$_{cc}$ keeps the 5000 most recently generated conflict clauses and all conflict clauses with no more than 50 literals.

Answer Set Solvers. In the absence of disjunctive rules (as with Smodels), the heart of an answer set solver is a search for stable models for normal logic programs. Currently the most frequently cited are Smodels and DLV. The code for Smodels$_{cc}$ is a modification of the code for Smodels, which is open source software. Accordingly, Smodels$_{cc}$ is also open source.[3]

Smodels replaces the simple unit propagation inference rule of DPLL SAT solvers with a set of inference rules based upon an inflationary variant of the wellfounded semantics; oversimplifying, we shall refer to this as closing under the wellfounded semantics. After closing under the wellfounded semantics, Smodels computes a *lookahead* on each unassigned variable x — called a *unit lookahead*: Assuming that x is *true* (*resp.*, *false*), it computes the wellfounded extension. If that gives a contradiction, Smodels infers that x is false (*resp.*, true). If it infers both, it backtracks; otherwise, it branches on (next guesses) a literal λ maximizing the inferences obtained by looking ahead on λ and $\neg\lambda$.

Smodels does not use conflict clauses or restarts.

Reducing Answer Sets to CNF-SAT. As noted above, we restrict attention to answer set solvers which can handle non-tight programs, even though solvers restricted to tight programs may be highly useful. (Indeed many frequently cited "benchmarks," such as graph coloring, naturally translate to tight programs.)

ASSAT and Cmodels-2 are general purpose Answer Set solvers that call SAT solvers to do most of their work. Given a program P, they pass the program completion $\overline{P}$ to a SAT solver. If $\overline{P}$ has no model, P has no stable model, and ASSAT and Cmodels-2 report "no."

A set U of atoms is *unfounded* over a partial truth assignment A if, for every rule $a \leftarrow b_1, \ldots, b_k, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m$ of P with $a \in A$, either (i) some $\neg b_i$ or $c_j \in A$ or (ii) some $b_i \in U$. If U is unfounded, the stable and well-founded semantics infer $\{\neg a : a \in U\}$, as a form of negation as failure. The program completion achieves only part (i) above, inferring fewer negative literals.

[2] Since the sets of conflict clauses may be huge, traversing all of C to do unit propagation is prohibitively slow. The best current method to do this seems to be Chaff's "optimized Boolean Constraint Propagation" [MMZZM01]. We use that also in our program.

[3] Smodels$_{cc}$, and the benchmarks used in this paper, can be obtained at http://www.ececs.uc.edu/~ schlipf/.

If the SAT solver returns a model, ASSAT and Cmodels-2 check whether all atoms in all unfounded sets are assigned false. If so, they return the model. If not, they create "loop formulas," which exclude the model, and feed them back into the SAT solver — repeating until a stable model is found or the search fails. The alternation between the SAT-solver phase and the unfounded-set-check phase may be inefficient, since the SAT solver may spend a great deal of time searching a branch which immediately could be pruned away by detecting an unfounded set.

An important advantage of ASSAT and Cmodels-2 is a sort of modularity: they are free to adapt to whatever SAT solver proves to be experimentally best for their applications. Smodels$_{cc}$ merges the "classical inference" part with the unfounded set check and thus sacrifices this modularity.

3 Conflict Clause Generation

Crucial to many modern SAT solvers is creating, storing, and looking up conflict clauses. Zhang *et al.* [ZMMM01] studies several different strategies for generating conflict clauses, implementing each of them in the zChaff variant of Chaff. We describe here their most effective strategy (the *1UIP* strategy) and how we adapted it for Smodels$_{cc}$.

When a solver such as Chaff detects a contradiction, it does a "critical path analysis" to choose a conflict clause. Because Chaff's inferences are derived through unit propagation only, if x and $\neg x$ are both inferred, they must have been inferred *since the latest choice made by the brancher*. Suppose that λ_0 was the last choice of the brancher. Chaff reconstructs the sequence of inferences used to infer each truth assignment since that last choice, representing it with a digraph, called the *implication graph*. The nodes are literals. There is an edge from literal λ_1 to λ_2 if a clause $\{\lambda_2, \neg\lambda_1, \ldots\}$ was used to infer λ_2. See Fig. 1.

So there is at least one directed path in the implication graph from λ_0 to x, and at least one from λ_0 to $\neg x$. A node on all these directed paths is called a *unique implication point* (*UIP*). Literal λ_0 itself is a UIP. All UIPs must lie on a single path from λ_0. Pick the UIP λ' farthest from λ_0 (i.e., closest to the contradiction). The derivation of the contradictory x and $\neg x$ now can be broken into (i) a derivation of λ' from λ_0 plus (ii) a derivation of the contradiction from λ'. By choice of λ', clauses involved in part (ii) contain only λ' plus some literals $\kappa_1, \ldots, \kappa_m$ that had been guessed or derived before Chaff branched on λ_0 — from a point higher up the search stack. The new conflict clause is $\neg\kappa_1 \vee \neg\kappa_2 \vee \cdots \vee \neg\kappa_m \vee \neg\lambda'$; for the example in Fig. 1, that lemma is $\neg\kappa_1 \vee \neg\kappa_2 \vee \neg\kappa_3 \vee \neg\kappa_4 \vee \neg\kappa_5 \vee \neg\kappa_{10} \vee \neg\kappa_{11} \vee \neg\kappa_{12} \vee \neg\lambda'$.[4]

At this point, Chaff does not simply backtrack on the last choice assignment. Rather, it "backjumps" to the level in the search tree where the last κ_i was guessed or inferred and restarts the search there with the new conflict clause in the cache. It will infer $\neg\lambda'$ at that level before going on with the search; this will keep it from exactly retracing its previous chain of guesses and inferences.

In a stable model solver, such as Smodels, contradictory literals need not be inferred in the same level of the backtrack search: an atom x may be inferred by forward or

[4] It is also possible to store multiple conflict clauses per contradiction. Following [ZMMM01], we create only one conflict clause for each contradiction.

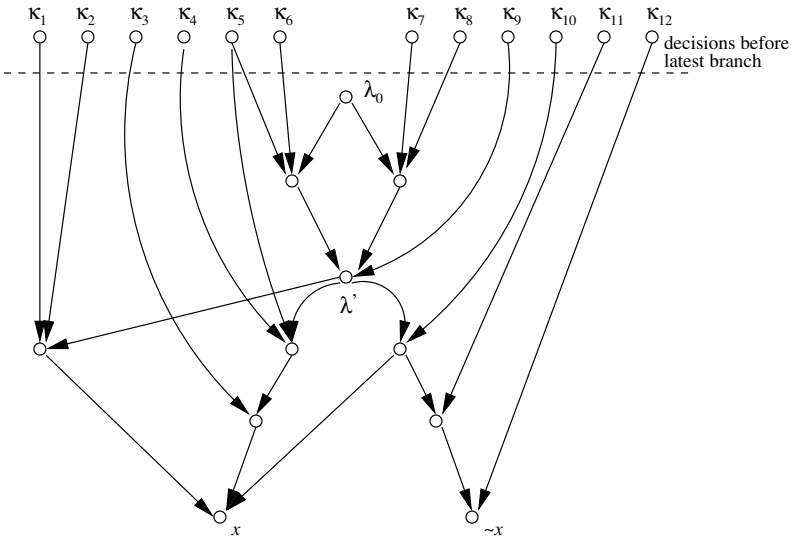


Fig. 1. An implication graph for a contradiction

contrapositive reasoning at one level, while x may appear in an unfounded set earlier or later. Thus in Smodels$_{cc}$, the construction above is altered a little. The *later* of $x, \neg x$ to be guessed or inferred is called the *conflict literal*, and, for the purposes of finding a UIP, an extra edge is added into the digraph, from the earlier literal in the conflicting pair to the conflict literal. Now a UIP is defined to be a literal, other than the conflict literal itself, appearing on every path from the guessed literal λ_0 to the conflict literal. Identifying the UIPs in Smodels$_{cc}$ is complicated by the fact that the implication graph may have cycles (in the case of inferences based upon unfounded sets). Nonetheless, identifying the UIPs may be accomplished in $O(|E|)$ time, where E is the set of edges incident to vertices between the choice literal and the conflict literal. Otherwise, Smodels$_{cc}$ constructs its conflict clauses from the implication graph as described above.

Smodels$_{cc}$ uses Smodels' five different inference rules (below), four corresponding to unit propagation on the completion of the program, and the fifth an unfounded set rule. For each of these inference rules we describe below how we add corresponding edges to the implication graph. Compared to construction of the implication graph in a DPLL-based SAT solver, implication graph construction in Smodels$_{cc}$ is relatively complex and costly.

Forward inference. *If all the subgoals in a rule*

$$a \leftarrow b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m$$

are true in the current assignment, infer a. Add edges from all b_i's and $\neg c_j$'s to a in the implication graph.

Kripke-Kleene negation (*all rules canceled*): *If every rule with head* a *has at least one subgoal negated in the current truth assignment, infer* $\neg a$. For each rule $a \leftarrow$

$b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m$ with head a, determine the cancelling assignment, $\neg b_i$ or c_j, which was guessed or inferred first (at the earliest level of the backtracking search), and add an edge from that assignment to $\neg a$ in the implication graph.

Contraposition for true heads. *If atom a is true in the current truth assignment, and if every rule with head a* except one *has at least one subgoal that is false in the current truth assignment, infer all the subgoals of that remaining rule to be true.* For example, suppose the only rules with a in their heads are $a \leftarrow b, c, not\ d$; $a \leftarrow e, f$; and $a \leftarrow not\ g, h$; and that the current truth assignment contains $a, d, \neg e$. Then $\neg g, h$ will be inferred. Add edges from each of $a, d, \neg e$ to each of $\neg g, h$ into the implication graph.

Contraposition for false heads. *If an atom a is false in the current truth assignment and some rule $a \leftarrow \lambda_1, \ldots, \lambda_k$ has every λ_i except one true in the current truth assignment, infer that last λ_i to be false.* For example, suppose the rule is $a \leftarrow b, c, not\ d$ and that $\neg a, b, c$ are in the truth assignment. Infer d, and add edges from each of $\neg a, b, c$ to d.

Wellfounded negation (Smodels' *at most*): *Temporarily removing all satisfied and undefined negative subgoals in all rules of the program yields a Horn program. Compute its least model M; the set of atoms* false *in M is unfounded; set these atoms to false in the current partial assignment*. (This is the logic; Smodels has a more complicated but faster calculation.)

For example, suppose that P contains the rules $a \leftarrow b$; $b \leftarrow c$; $c \leftarrow a$; and $a \leftarrow d$; and that these are the only rules with a, b, or c in their heads. Suppose also that $\neg d$ is in the current partial truth assignment. Then infer $\neg a$, $\neg b$, and $\neg c$. Add edges from $\neg d$ to $\neg a$, from $\neg a$ to $\neg c$, from $\neg c$ to $\neg b$, and from $\neg b$ to $\neg a$. Note that here the implication graph contains a cycle.

The algorithm for determining the edges for wellfounded negation is similar to the one for Kripke-Kleene negation: If an atom a has been detected to be unfounded then it will be the case that Smodels has found a reason to cancel every rule R with a at the head. As in Kripke-Kleene negation, add an edge from the earliest reason for the cancellation to the literal $\neg a$. In particular, if the earliest reason is that a set U of atoms mentioned positively in the body of R have become simultaneously unfounded (unsupported) with a, then the edge will have as its source node $\neg x$ where x is the member of U which was first permanently removed from Smodels' set of supported atoms during the current unfounded set check.

4 Search Heuristics

As noted earlier, Smodel's search heuristic is based on its unit lookaheads. Chaff weights its literals based upon how many of the conflict clauses they occur in[5] and always branches on an unassigned literal of maximum weight; thus Chaff can be thought of as "learning key literals."

The heuristic used in Smodels$_{cc}$ is modeled after the one in BerkMin. It works as follows: Each variable x has an "activity count", $ac(x)$, which counts the number of

[5] It also lets the weights from older clauses decay with time.

times that either x or $\neg x$ has been involved in producing a conflict (i.e., has appeared in a conflict clause or has appeared in an implication graph along a path from a conflict node to a variable in the corresponding conflict clause). We choose the branching literal x_0 or $\neg x_0$ such that $ac(x_0)$ is maximized, with a restriction: If there are unsatisfied conflict clauses in the cache, then x_0 must be chosen from the most recently constructed unsatisfied conflict clause. Branch on x_0 or $\neg x_0$, whichever has appeared in more conflict clauses.[6]

5 Experimental Results

We performed experiments to compare the performance of Smodels (version 2.26), ASSAT (version 1.50), Cmodels-2 (version 1.04), and Smodels$_{cc}$. Our experiments were run on 1533 Mhz Athlon XP processors with 1 GB of main memory, using Linux. With ASSAT, we used the -s 2 command line option, which seemed to give marginally better results than the default or the -s 1 settings. For Cmodels, we used the default settings, which produced the best results on tight problems. However, on Hamiltonian cycle problems, we used Cmodels' -si setting, which produced significant performance improvements in that domain. The SAT solver used by ASSAT was Chaff2 (version spelt3) [Chaff2], which is ASSAT's default solver. Cmodels used the zChaff SAT solver (version 2003.7.1) for tight problems and SIMO (version 3.0) [SIMO] for Hamiltonian cycle problems, as dictated by the command line settings. Note that ASSAT and Cmodels benefit from conflict clauses in our tests, because conflict clauses are heavily incorporated into Chaff and SIMO. In each of the tables below (except for the DLX benchmarks), the run times given represent the median and maximum number of user time seconds taken to solve each of eleven (11) randomly generated problem instances. Each search process was aborted ("timed out") after 3600 seconds. Runtimes reported include the time to execute Lparse, the default grounder for Smodels, ASSAT, Cmodels, and Smodels$_{cc}$.

We concentrated on three problem domains:

Boolean satisfiability. Our tests in this domain include some randomly generated 3-SAT problems and 16 of the "DLX" circuit verification benchmarks from Miroslav Velev [VB99]. [7] In each case, the problem is provided as a CNF-SAT problem, which we have converted to an answer set program. We do not expect to outperform ASSAT or

[6] We also tested a version of Smodels$_{cc}$ which used lookaheads for the search heuristic (Smodels' original default heuristic). This version frequently had somewhat smaller search trees than did Smodels$_{cc}$ with the BerkMin-like heuristic (possibly because of the extra pruning afforded when contradictions are found during lookaheads). It solved a substantial number of our experimental test problems much faster than did the original Smodels, showing that much of the power of conflict clauses comes from the backjumping and unit propagation-based pruning which they afford (i.e., not only from how they affect the search heuristic). However, because the BerkMin-like heuristic is so much less expensive to compute than the lookahead-based heuristic, we found that it provided substantially better overall performance in our experiments. Thus, all of the timings for Smodels$_{cc}$ in our Experimental Results section were obtained using the BerkMin-like heuristic.

[7] The benchmarks which we used were from the Superscalar Suite 1.0 (SSS.1.0), available at http://www.ece.cmu.edu/~ mvelev. The eight satisfiable instances which we tested were

Cmodels on these examples since the logic programs are tight. Trying the "dlx" problems was an attempt to show that conflict clauses can be particularly helpful on non-uniform, "real world" data.

Median and maximum seconds on 11 random 3-SAT problems

Data set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
vars	clauses	median	max	median	max	median	max	median	max	
250	1025	16.5	112.2	9.0	891.0	3.3	552.2	2.5	746.0	10
250	1050	86.1	207.2	1056.8	2136.4	334.3	2321.1	777.5	1761.2	6
250	1075	133.4	376.6	576.4	1 abort	801.1	2 abort	883.9	1 abort	3
250	1100	74.4	169.8	432.8	2110.8	329.9	1 abort	360.0	2216.0	1

Median and maximum seconds on 8 DLX benchmark problems

Data set	Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
	median	max	median	max	median	max	median	max	
8 satisfiable	>3600	8 abort	11.6	17.9	2.6	4.4	2.9	10.8	8
8 unsatisfiable	>3600	7 abort	15.9	41.4	6.8	18.2	8.6	14.4	0

Graph k-coloring problems. We study these because they are standard in the literature. Since the program is tight, much of the sophistication of answer set programming is not needed. Nevertheless, it is important that the solver be reasonably efficient on such problems.

We first generated problems on uniform, random graphs with 400 to 500 vertices. We also generated "clumpy" problems in this domain by making graphs of 100 clumps with 100 nodes each (for a total of 10,000 nodes per graph). For the first set of clumpy graphs we set our density parameter $d = 150$, which means that we randomly placed 150 edges in each clump and 150 edges between clumps. This gave us graphs with a total of $100 \times 150 + 150 = 15{,}150$ edges each. We then increased d to 170 and 190, obtaining graphs with 17,170 and 19,190 edges, respectively.

As with Boolean satisfiability, we expected ASSAT and Cmodels to outperform Smodels$_{cc}$ since the program is tight. We expected Smodels to be faster than Smodels$_{cc}$ on fairly "uniform" graphs, and Smodels$_{cc}$ to be faster than Smodels on "non-uniform" graphs.

Hamiltonian cycle problems. Among common current benchmarks, these may be the "gold standard" for answer set solvers since the problem description is not tight. We used directed graphs in these experiments.

Here we considered three reductions to answer set programming. The first, a standard reduction frequently used in benchmarking, was taken from [Nie99].[8] The second reduction was a "tight on its models" reduction used with ASSAT in [LZ03]. The third reduction is the modification below of the first:

dlx2_cc_bug01.cnf, ..., dlx2_cc_bug08.cnf The eight unsatisfiable instances were dlx1_c.cnf, dlx2_aa.cnf, dlx2_ca.cnf, dlx2_cc.cnf, dlx2_cl.cnf, dlx2_cs.cnf, dlx2_la.cnf, and dlx2_sa.cnf.

[8] Except that the extended rules were replaced with choice constructions since ASSAT and Smodels$_{cc}$ do not currently support the extended rules. They will be added to Smodels$_{cc}$ soon.

Median and max secs on 11 random 3-coloring problems, uniform distribution of edges

Data set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
vertices	edges	median	max	median	max	median	max	median	max	
400	900	3.4	163.0	1.9	197.2	0.3	88.3	0.3	88.2	10
400	950	99.5	511.8	89.8	1704.4	58.3	837.7	24.6	837.6	2
400	1000	20.7	134.3	16.0	68.4	3.7	14.3	4.0	13.1	0
500	1100	0.8	4.2	1.4	6.4	0.2	2.4	0.3	2.2	11
500	1150	196.0	356.6	697.9	2 abort	44.4	2 abort	188.3	2 abort	10
500	1200	2753.9	5 abort	>3600	8 abort	>3600	8 abort	>3600	8 abort	0-5

Median and max secs on 11 random 3-coloring problems, clumpy distribution of edges

Data set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
vertices	edges	median	max	median	max	median	max	median	max	
10000	15150	256.1	1 abort	21.3	50.5	8.0	9.7	8.4	10.1	10
10000	17170	201.6	3 abort	19.2	21.7	8.5	11.4	10.1	11.8	7
10000	19190	>3600	8 abort	8.5	223.4	3.2	12.9	4.2	11.7	2

```
hc(X,Y) :- not not_hc(X,Y), edge(X,Y).
not_hc(X,Y) :- not hc(X,Y), edge(X,Y).
:- hc(X1,Y), hc(X2,Y), edge(X1,Y), edge(X2,Y), vtx(Y), X1 != X2.
:- hc(X,Y1), hc(X,Y2), edge(X,Y1), edge(X,Y2), vtx(X), Y1 != Y2.
:- vtx(X), not r(X).
r(Y) :- hc(X,Y), edge(X,Y), initialvtx(X).
r(Y) :- hc(X,Y), edge(X,Y), r(X), not initialvtx(X).
outgoing(V) :- edge(V,U), hc(V,U).     % added in 3rd reduction
:- vtx(V), not outgoing(V).            % added in 3rd reduction
```

The two lines which we added to obtain the third reduction state explicitly that, in a Hamiltonian cycle, every node must have an outgoing edge. This fact is implicit in the reduction from [Nie99]. However, stating it explicitly helps the solvers prune their search spaces. We note that these two lines were included in the "tight on its models" reduction from [LZ03].

In our experiments the third reduction was always faster than the first two, so we used it in all experiments reported here.[9] In all of these experiments we enforced the restriction that every vertex in every graph must have an in-degree and an out-degree ≥ 1 to avoid trivial examples.

We nonetheless found it difficult to generate hard Hamiltonian cycle problems using a random, uniform distribution of edges. For instance, our randomly generated problems with 6000 nodes were in all cases too large to run under ASSAT, which was understandable because the ground instantiations produced by Lparse were around 12MB in size.

[9] This performance improvement was very pronounced with Smodels and Smodels$_{cc}$. Using ASSAT the runtimes were only slightly better with the third reduction than with the second, but on larger problems, under the second reduction, the groundings provided by Lparse were too large for any of the solvers to handle. For example, on a randomly generated graph with 1000 nodes and 3500 edges, Lparse generated a ground instantiation of about 123MB under the second reduction. By contrast, the ground instantiation produced for the third reduction was 1.09MB, and the problem was easily solved by all of the solvers.

Yet these problems were not particularly difficult to solve, at least for Smodels$_{cc}$. Unsatisfiable instances taken from this distribution were generally solved by each of the solvers with no backtracks.

We sought to produce some hard Hamiltonian cycle problems that were of reasonable size and had a less uniform (more "clumpy") distribution of edges. (It is commonly believed that less uniform distributions are in fact common in actual applications.) For these experiments we randomly generated "clumpy" graphs which would be forced to have some, but relatively few, Hamiltonian cycles each.

For this purpose we generate a random mn-node "clumpy" graph G as follows: Let n be the number of clumps in the graph and let m be the number of nodes in each clump. First generate an n-node graph A as the "master graph," specifying how the clumps are to be connected; each vertex of A will correspond to a clump in the final graph. Add random edges to A until A has a Hamiltonian cycle.

Now generate the clump C corresponding to a vertex v of A. Let $x = indegree(v)$ and $y = outdegree(v)$. C has m nodes; select x nodes to be "in-nodes" and y different nodes to be "out-nodes"; increase C to $x + y$ nodes if $x + y > m$. Add random edges to clump C until there are Hamiltonian paths from each in-node of C to each out-node of C. (Thus C will have at least xy Hamiltonian paths.)

Finally, for every edge (v_i, v_j) in the master graph A, insert an edge from an out-node of C_i to an in-node of C_j. Every in-node in every clump is to have exactly one incoming edge from another clump. Likewise, every out-node in every clump is to have exactly one outgoing edge to another clump. G will have at least one Hamiltonian cycle.

Median and max secs on 11 Hamiltonian cycle problems, uniform distribution of edges

Data set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
vertices	edges	median	max	median	max	median	max	median	max	
1000	4000	1.0	1.1	0.9	1.1	0.9	1.1	1.0	1.1	0
1000	4500	1.2	132.0	1.2	3.6	1.1	971.3	1.3	308.0	4
1000	5000	172.7	201.0	3.6	4.8	52.6	1064.1	6.8	438.4	6
1000	5500	244.6	269.5	4.5	6.9	275.5	1440.6	141.1	175.7	10
6000	30000	8.4	3 abort	8.7	64.9	...	...	8.3	2 abort	3
6000	33000	>3600	9 abort	57.9	77.7	...	...	>3600	9 abort	9
6000	36000	>3600	7 abort	64.3	81.2	...	...	1963.4	3 abort	7
6000	39000	>3600	10 abort	78.4	109.9	...	...	>3600	7 abort	10

Median and max secs on 11 Hamiltonian cycle problems, clumpy distribution of edges

Data set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels		#Sat
# of clumps	vertices / clump	median	max	median	max	median	max	median	max	
10	10	0.4	1 abort	0.2	0.5	0.8	1.4	0.5	0.9	11
12	12	620.9	4 abort	0.7	1.1	2.6	10.8	2.6	4.5	11
14	14	>3600	10 abort	1.3	3.9	108.8	194.1	11.4	83.4	11
16	16	>3600	9 abort	6.5	40.5	263.0	608.8	24.8	100.6	11
18	18	>3600	11 abort	42.7	353.4	>3600	6 abort	133.7	2063.5	11

6 Conclusions

We believe that this study has confirmed the following two points:

Adding conflict clauses to answer set search will significantly increase speed on non-uniform problems. Experience has shown that conflict clause learning has immensely speeded up SAT solvers on "real world" data. We adapted conflict clause analysis to the answer set programming domain, notably by finding a reasonable way to represent inference by wellfounded negation. We tested this on some "real world" tight problems plus some randomly generated non-uniform problems. Smodels$_{cc}$ was consistently faster than Smodels, confirming our prediction. Interestingly, for uniform Hamiltonian Cycle problems, Smodels$_{cc}$ was also much faster than Smodels.

For non-tight programs, separating the classical analysis of the completion from the unfounded set check, as in ASSAT and Cmodels, is less efficient than merging the two processes into a single search. The obvious explanation seems to be that, with ASSAT or Cmodels, the SAT solver spends a great deal of time on parts of the search tree that an unfounded set check could eliminate early.

A significant advantage of ASSAT and Cmodels is that they can incorporate the latest, highly optimized SAT solvers with relatively little additional programming effort.

For future work, a useful middle ground between their approach and that of Smodels$_{cc}$ might be to run a state-of-the-art SAT solver on the program completion, but modify the SAT solver so that it includes an unfounded set check during the search.

Acknowledgement. This research was partially supported by U.S. DoD grant MDA 904-02-C-1162.

References

[BEL00] Babovich, Y., E. Erdem, and V. Lifschitz. Fage's Theorem and Answer Set Programming. *Proc. Int'l Workshop on Non-Monotonic Reasoning*, 2000.

[BM03] Babovich, Y. and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. Available from *http://www.cs.utexas.edu/users/tag/cmodels.html*.

[BS97] Bayardo, R. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. *Proc. of the Int'l Conf. on Automate Deduction*, 1997.

[Clark78] Clark, K. Negation as failure. In Herve Gallaire and Jack Minker, eds., *Logic and Data Bases*, 293–322, Plenum Press, 1978.

[DLL62] Davis, M., G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the Association of Computing Machinery* **5** (1962) 394–397.

[EFLP00] Eiter, T., W. Faber, N. Leone, G. Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, ed., *Logic-Based Artificial Intelligence* 2000, 79–103.

[Fages94] F. Fages. Consistency of Clark's completion and existence of stable models. *J. Methods of Logic in Computer Science* 1 (1994) 51–60.

[GMT03] Giunchiglia, E., M. Maratea, and A. Tacchella. Look-Ahead vs. Look-Back techniques in a modern SAT solver. SAT2003, May 5–8 2003. Portofino, Italy.

[GN02] Goldberg, E. and Y. Novikov. BerkMin: a Fast and Robust Sat–Solver. *Design Automation and Test in Europe (DATE)* 2002, 142–149.

[LZ03] Lin, F. and Z. Jicheng. On Tight Logic Programs and Yet Another Translation from Normal Logic Programs to Propositional Logic. *Proc. of IJCAI-03*. To appear, 2003.

[LZ02] Lin, F. and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Proc. of the 18th Nat'l. Conf. on Artificial Intelligence (AAAI-2002)*.

[MS99] Marques-Silva, J.P. and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers* 48(5), 1999, 506–521.

[MMZZM01] Moskewicz, M., C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. *Proc. of the 38th Design Automation Conf. (DAC'01)*, 2001.

[NSS00] Niemela, I., P. Simons, and T. Syrjanen. Smodels: a system for answer set programming. *Proc. of the 8th Int'l Workshop on Non-Monotonic Reasoning* 2000.

[Nie99] Niemela, I. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3–4) 1999, 241–273.

[Sim00] Simons, P. Extending and Implementing the Stable Model Semantics. PhD dissertation, *Helsinki University of Technology* 2000.

[VB99] Velev, M., and R. Bryant. Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic. *Correct Hardware Design and Verification Methods (CHARME'99)* 1999.

[Zheng97] Zhang, H. SATO: An Efficient Propositional Prover. *Proc. of Int'l Conf. on Automated Deduction* 1997.

[ZMMM01] Zhang, L., C. Madigan, M. Moskewicz, and S. Malik.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. *Proc. of ICCAD 2001*.

[ASSAT] ASSAT code available from *http://assat.cs.ust.hk/*.

[Cmod] Cmodels code available from *http://www.cs.utexas.edu/users/tag/cmodels.html*.

[Chaff2] Chaff2 code available from *http://www.ee.princeton.edu/ chaff/index1.html*.

[SIMO] Simo code is included in the Cmodels-2 distribution. See also *http://www.mrg.dist.unige.it/ sim/simo/*.

Properties of Iterated Multiple Belief Revision

Dongmo Zhang

School of Computing and Information Technology
University of Western Sydney, Australia
dongmo@cit.uws.edu.au

Abstract. In this paper we investigate the properties of iterated multiple belief revision. We examine several typical assumptions for iterated revision operations with an ontology where an agent assigns ordinals to beliefs, representing strength or firmness of beliefs. A notion of minimal change is introduced to express the idea that if no evidence to show how a belief set should be reordered after it is revised, the changes on the ordering should be minimal. It has been shown that under the assumption of minimal change, the multiple version of Darwiche and Pearl's postulate (C1) holds no matter in what degree new information is accepted. Moreover, under the same assumption, Boutilier's postulate (CB) holds if and only if new information is always accepted in the lowest degree of firmness while Nayak *et al.*'s postulate (CN) holds if and only if new information is always accepted in the highest degree. These results provide an ontological base for analyzing the rationality of postulates of iterated belief revision.

Keywords: Iterated belief revision, multiple belief revision, belief revision.

1 Introduction

Iterated belief revision has been intensively investigated in the community of belief revision[2][5][9][10][13][17]. The major concern in the research is the reducibility of iterated revisions to single step revisions. Boutilier in [2] proposed an assumption to capture the relationship, named (CB) by Darwiche and Pearl in [4]:

(CB) if $K * A \vdash \neg B$, then $(K * A) * B = K * B$

Darwiche and Pearl in [4] argued that (CB) would be overcommitted and could be weakened and enhanced elsewhere with the following assumptions:

($C1$**)** If $B \vdash A$, then $(K * A) * B = K * B$.
($C2$**)** If $B \vdash \neg A$, then $(K * A) * B = K * B$.
($C3$**)** If $A \in K * B$, then $A \in (K * A) * B$.
($C4$**)** If $\neg A \not/\in K * B$, then $\neg A \not/\in K * A) * B$.

It was shown by Freud and Lehmann (see [10]), however, that these postulates do not go well with AGM framework: $(C2)$ *is inconsistent with AGM postulates*. Nevertheless, Darwiche and Pearl argued in [5] that the original AGM postulates should be weakened in order to accommodate the additional postulates for iterated revision. More other iterated belief revision frameworks have been also proposed in the last few

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 314–325, 2004.

years, reflecting different philosophy of iterated belief revision[9][10][14]. No matter how much divergency these frameworks have made it has been generally accepted that belief change should not be considered as a purely set-theoretical change of belief sets but an evolution of epistemic states, which encapsulate beliefs with the information of firmness of beliefs. The change of epistemic states involves not only the change on belief set but also the change on orderings over the belief set. Whenever a belief set is revised by new information, the ordering of the belief set should also change to accommodate the new information. The further revision will be based on the new ordering. Therefore the posterior revision operation is normally not identical with the prior operation. This idea has been explicitly expressed by Nayak *et al.* in [14] by using a subscription to differentiate two steps revision. For instance, $(C1)$ and $(C2)$ can be restated in the following form:

$(C1')$ If $B \vdash A$, then $(K * A) *_A B = K * B$.
$(C2')$ If $B \vdash \neg A$, then $(K * A) *_A B = K * B$.[1]

We remark that this change is significant since the postulates will no long lay restrictions on single step revisions but to specify the relationship between the first step revision and the second step revision. In fact, it has been shown in [14] that $(C1')$ and $(C2')$ are consistent with AGM postulates. Therefore we can use these additional postulates to extend the AGM framework without revising the AGM postulates. Moreover, Nayak *et al.* in [14] even showed that $(C1')$ can be strengthened further by the following postulate without loss of consistency:

(CN) If $A \wedge B \nvdash \bot$, then $(K * A) *_A B = K * (A \wedge B)$.

Most of the research in iterated revision based on single belief revision, that is, the new information is represented by a single sentence. It is even more interesting if we consider the problem in the setting of multiple belief revision. By using the notation introduced in [22] and Nayak *et al.*'s notation, all the above postulates can be restated in the following form:

($\otimes C1$) $(K \otimes F_1) \otimes_{F_1} (F_1 \cup F_2) = K \otimes (F_1 \cup F_2)$.
($\otimes C2$) If $F_1 \cup F_2$ is inconsistent, then $(K \otimes F_1) \otimes_{F_2} F_2 = K \otimes F_2$.
($\otimes C3$) If $F_1 \subseteq K \otimes F_2$, then $F_1 \subseteq (K \otimes F_1) \otimes_{F_1} F_2$.
($\otimes C4$) If $F_1 \cup (K \otimes F_2)$ is consistent, then $F_1 \cup ((K \otimes F_1) \otimes_{F_1} F_2)$ is consistent.
($\otimes CB$) If $F_2 \cup (K \otimes F_1)$ is inconsistent, then $(K \otimes F_1) \otimes_{F_1} F_2 = K \otimes F_2$.
($\otimes CN$) If $F_1 \cup F_2$ is consistent, $(K \otimes F_1) \otimes_{F_1} F_2 = K \otimes (F_1 \cup F_2)$.

Surprisingly, the extension does not increase the complexity of these postulates. Contrarily, they become more readable. For instance, $(\otimes C1)$ expresses the assumption that *if the second revision confirms all the information contained in the first one, the first revision is superfluous*[10].

One question is to be answered that whether these postulates are consistent with the multiple version of AGM postulates, especially with the Limit Postulate introduced by Zhang *et. al.* in [19]. Another question, which is more important, that

[1] In [14] an extra condition $\nvdash \neg A$ is added to avoid the treatment of inconsistent belief set.

what are the underlying ontology to use these postulates. As Friedman and Halpern pointed out in [6] that there has been too much attention paid to postulates and not enough to the underlying ontology. We need to make it clear that *in what situation an agent revises her epistemic states in the way that we specified by these postulates*.

In this paper we exploit an ontology where an agent assigns an ordinal to each of her belief, representing the strength or firmness of the belief[2]. We introduce a notion of minimal change of belief degrees, which expresses the idea that if no evidence to show that how a belief set should be ordered after it is revised, the change of the ordering on the belief set should be minimal. We show that under the assumption of minimal change of ordering, $(\otimes C1)$ holds no matter in what degree the new information is believed. However, the postulates $(\otimes CN)$ and $(\otimes CB)$ heavily depend on how the new information is accepted. We show that $(\otimes CN)$ holds if and only if the new information is accepted and kept in the highest degree of firmness whereas $(\otimes CB)$ holds if and only if the new information is accepted in the lowest degree comparing with the old beliefs. These results provide an ontological base for the analysis of the rationality on the postulates for iterated belief revision.

Throughout this paper, we consider a propositional language $\mathcal{L}$ as the object language. We denote individual sentences in $\mathcal{L}$ by A, B, or C, and denote sets of sentences by F,F_1, F_2 etc. If F is an infinite set of sentences, $\bar{F}$ denotes a finite subset of F. If F is a finite set, $\wedge F$ means the conjunction of its elements. We shall assume that the underlying logic includes the classical first-order logic with the standard interpretation. The notation $\vdash$ means the classical first-order derivability and Cn the corresponding closure operator, i.e.,

$$A \in Cn(\Gamma) \text{ if and only if } \Gamma \vdash A$$

A set K of sentences is a belief set if $K = Cn(K)$.

2 Preliminaries in Multiple Belief Revision

Firstly, let's review the basic concepts in mutual belief revision. Zhang and Foo in [22] introduced a version of multiple belief revision, called *set revision*, which allows to revise a belief set by any set of sentence or another belief set. Nine postulates were proposed. Among them the first eight postulates are the direct generalization of the associated AGM ones(also see [12][13]). The last one, called *Limit Postulate*, deals with the compactness of infinite belief revision [22], which says that a revision by an infinite belief set can be approached by the revisions of the belief set with its finite subsets.

Formally, let $\mathcal{K}$ be the set of all belief sets. A function $\otimes$ is called a *set revision function* if it satisfies the following postulates:

$(\otimes 1)$ $K \otimes F = Cn(K \otimes F)$.
$(\otimes 2)$ $F \subseteq K \otimes F$.

[2] In fact, the ordinal will be interpreted as the strength of disbelief due to technical reason.

($\otimes$3) $K \otimes F \subseteq K + F$.
($\otimes$4) If $F \cup K$ is consistent, then $K + F \subseteq K \otimes F$.
($\otimes$5) $K \otimes F$ is inconsistent if and only if F is inconsistent.
($\otimes$6) If $Cn(F_1) = Cn(F_2)$, then $K \otimes F_1 = K \otimes F_2$.
($\otimes$7) $K \otimes (F_1 \cup F_2) \subseteq (K \otimes F_1) + F_2$.
($\otimes$8) If $F_2 \cup (K \otimes F_1)$ is consistent, then $(K \otimes F_1) + F_2 \subseteq K \otimes (F_1 \cup F_2)$.
($\otimes LP$) $K \otimes F = \bigcup_{\bar{F} \subseteq F} \bigcap_{\substack{\bar{F}' \subseteq_f Cn(F) \\ \bar{F} \subseteq \bar{F}'}} K \otimes \bar{F}'$.

where K is a belief set. F, F_1, F_2 are sets of sentences. $\bar{F} \subseteq_f F$ means $\bar{F}$ is a finite subset of F.

The model of set revision is based on the following variant of epistemic entrenchment.

Definition 1. [18] Let K be a belief set, $\mathcal{P}$ a partition of K, and $<$ a total order over $\mathcal{P}$. The triple $K = (K, \mathcal{P}, <)$ is called *a total-ordered partition(TOP)* of K. For any $P \in \mathcal{P}$ and $A \in P$, P is called the *rank* of A, denoted by $r(A)$.

A NOP is called a *nicely-ordered partition* (NOP) if it satisfies the following *Logical Constraint*:

If $A_1, \cdots, A_n \vdash B$, then $r(B) \leq \max\{r(A_1), \cdots, r(A_n)\}$.

It is easy to see that a TOP is nothing but a total pre-order on belief set. The ordering $<$ in an NOP is essentially the reverse order of epistemic entrenchment(EE). Logical Constraint here is the combination of (EE2) and (EE3)[7]. (EE5) does not satisfied by NOP. Therefore NOP is weaker version of EE (see more detail about the relationship between NOP and EE in [22]).

With the notion of NOP, a set revision operator can be constructed as follows:

Definition 2. [19] Let $\Sigma = (K, \mathcal{P}, <)$ be a NOP of a belief set K. Define a revision function $\otimes$ as follows: for any set F of sentences,

i). If $F \cup K$ is consistent, then $K \otimes F = K + F$; otherwise,

ii). $B \in K \otimes F$ if and only if there exists $A \in K$ such that $F \vdash \neg A$ and

$$\forall C \in K((A \vdash C \ \& \ F \vdash \neg C) \Rightarrow (r(C \vee B) < r(C) \text{ or } \vdash C \vee B)) \qquad (1)$$

$\otimes$ is called an *NOP-based revision.*

It has been shown in [22] that *a set revision operator satisfies the nine postulates if and only if it is a NOP-based revision*[3].

3 Iterated Multiple Belief Revision

Now we consider iterated operations in the setting of multiple belief revision. Firstly, let's consider the relationship between the additional postulates.

[3] The original representation theorem was given for contraction operator. The representation result for revision operator can be easily obtained by using Levi Identity.

3.1 Relationship of Iterated Revision Postulates

The following lemma shows the relationship between the postulates for iterated revision and AGM postulates.

Lemma 1. *Let $\otimes$ be a revision function that satisfies ($\otimes$1)-($\otimes$6). Then*

i). ($\otimes C1$) implies ($\otimes 7$) and ($\otimes 8$).
ii). ($\otimes CB$) implies ($\otimes C1$) and ($\otimes C2$).
iii). ($\otimes CN$) implies ($\otimes C1$), ($\otimes C3$) and ($\otimes C4$).

Due to the relationship we will concentrate on the postulate $(\otimes C1)$, $(\otimes CB)$ and $(\otimes CN)$ only in the sequent of the paper. We will examine the conditions when the postulates hold. First, let's find an ontology where these conditions can be stated.

3.2 Perfectly-Ordered Partition

In [20], Zhang *et al.* introduced a special kind of nice-ordered partition in which the partition is well-ordered.

Definition 3. [20] A nicely-ordered partition $\Sigma = (K, \mathcal{P}, <)$ is called a *perfectly-ordered partition* (POP) if $<$ is a well-order on $\mathcal{P}$. An NOP-based revision function is a *POP-based revision* if it is generated by a perfectly-ordered partition.

Mapping a sentence to a natural number or an ordinal is one of the traditional ways to rank beliefs. Applying logical constraint on the mapping to make it to be an epistemic entrenchment ordering has been also exploited in [17][18]. However, the ordering we use here should be differentiated from the ways to map possible worlds to ordinals[16] or to map Spohn's system of spheres to ordinals[15] since the underlying ontologies are significantly different even though they might be interconvertible.

The following lemmas show the logical properties of a POP.

Lemma 2. *Let $\Sigma = (K, \mathcal{P}, <)$ be a POP and η the ordinal type of $<$. Let $P_\alpha = \{A \in K : r(A) = \alpha\}$. Let $P_{\leq\alpha} = \bigcup_{\beta\leq\alpha} P_\beta$. Then for any $\alpha < \eta$, $P_{\leq\alpha}$ is logically closed.*

Conversely, let r be a function mapping a belief set K to an ordinal η. If, for any $\alpha < \eta$, $\{A \in K : r(A) \leq \alpha\}$ is logically closed, then r determines a POP over K.

Lemma 3. *Let $\otimes$ be a revision function generated by a POP $\Sigma = (K, \mathcal{P}, <)$. For any set F of sentences, if $F \cup K$ is inconsistent, then $B \in K \otimes F$ if and only if there exists a sentence $A \in K$ such that $F \vdash \neg A$ and*

$$b(A \vee B) < P_{min} \text{ or} \vdash A \vee B$$

where $P_{min} = \min\{r(A) : A \in K \ \& \ F \vdash \neg A\}$.

3.3 Minimal Change of Belief Degrees

The theory of belief revision is dominated by the principle of minimal change. Such minimal change should not mean the change in cardinality of belief sets but mean the change of epistemic states: belief sets and associated degrees of beliefs. Such a distinction is not essential in one shot revision but becomes crucial in iterated revisions. An iterated revision operation should be viewed as a function with three inputs (*belief set, new information, ordering over the belief set*) and two outputs (*new belief set, new ordering*). The change of the ordering during revision partially determines the relationship between one step revision and iterated revision. Therefore different philosophy of minimal change of ordering leads to different postulates for iterated revision[2][13]. The following definition depicts an instantiation of the principle of minimal change that the ordering on beliefs should keep unchanged unless it violates the Logical Constraint.

Definition 4. Let $\otimes$ be a revision function based on a POP $\Sigma = (K, \mathcal{P}, <)$ and η be the order type of $\mathcal{P}$. For any set F of sentences and an ordinal α, define a POP $\Sigma^{F,\alpha} = (K \otimes F, \mathcal{P}^{F,\alpha}, <^{F,\alpha})$ over $K \otimes F$ as follows:

1. For any $\beta < \max\{\eta, \alpha + 1\}$,
   $P_\beta^{F,\alpha} = \begin{cases} P_\beta \cap (K \otimes F), & \text{if } \beta < \alpha; \\ (Cn(P_{\leq\beta} \cup F) \cap K \otimes F) \backslash P_{<\beta}^{F,\alpha}, & \text{otherwise.} \end{cases}$
   where $P_{\leq\beta} = \bigcup_{\gamma \leq \beta} P_\gamma$ and $P_{<\beta}^{F,\alpha} = \bigcup_{\gamma<\beta} P_\gamma^{F,\alpha}$.
2. Let $\mathcal{P}^{F,\alpha} = \{P_\beta^{F,\alpha} : \beta < \max\{\eta, \alpha + 1\}\}$.
3. For any $P_\beta^{F,\alpha}$, $P_\gamma^{F,\alpha} \in \mathcal{P}^{F,\alpha}$, define:

$$P_\beta^{F,\alpha} <^{F,\alpha} P_\gamma^{F,\alpha} \text{ if and only if } \beta < \gamma$$

We call $\Sigma^{F,\alpha}$ the *minimal change of Σ with respect to F and α*.

The intuition behind the definition is that if we accept the new information F with the degree of α, F and its logical consequence with K will be merged to the old partition in the way that the ordering of partition will keep unchange except the belief degree of some $K \cup F$'s logical consequence could be higher because them are enhanced by new information.

The following theorem shows that under the assumption of minimal change of POP, $(\otimes C1)$ holds no matter in what degree the new information is accepted.

Theorem 1. *Let $\otimes$ be a revision function based on a POP $\Sigma = (K, \mathcal{P}, <)$. Let $\otimes_{F_1}$ be the revision function based on a minimal change $\Sigma^{F_1,\alpha}$ of Σ. Then we have*

($\otimes C1$) $(K \otimes F_1) \otimes_{F_1} (F_1 \cup F_2) = K \otimes (F_1 \cup F_2)$.

Corollary 1. *($\otimes 1$)-($\otimes 6$) and ($\otimes LP$) are consistent with ($\otimes C1$).*

4 Two Radical Strategies of Accepting New Information

We have seen that, provided the change of belief degrees is minimal, $(\otimes C1)$ holds no matter where the new information is inserted. However, the strategy to accept new information significantly affects the result of iterated revision. In this section, we consider two radical strategies of accepting new information: extreme cases that new information is accepted in absolute affirmative and in extreme suspicion. The following theorem deals with the first case.

Theorem 2. *Let $\otimes$ be a revision function based on POP $\Sigma = (K, \mathcal{P}, <)$ and $\otimes_{F_1}$ the revision function based on Σ's minimal change $\Sigma^{F_1,\alpha}$. If $\alpha = 0$ and $P_0 \subseteq Cn(F_1)$, then we have*

($\otimes CN$) *If $F_1 \cup F_2$ is consistent, $(K \otimes F_1) \otimes_{F_1} F_2 = K \otimes (F_1 \cup F_2)$.*

The theorem shows that $(\otimes CN)$ holds if new information is always accepted in the top level of firmness(no weaker than any old beliefs). Surprisingly, the condition is not only sufficient but also necessary except for some limit cases. It is not hard to verify that $(\otimes CN)$ holds for any α if one of the following conditions is true:

1. $F_1 = Cn(\emptyset)$;
2. $P_0 \cup F_1$ *is inconsistent;*
3. *there is* β *such that* $P_{\leq\beta} = Cn(F_1)$.

By excluding these limit cases we have

Proposition 1. *If $F_1 \neq Cn(\emptyset)$, $P_\alpha \cup F_1$ is consistent and $P_\alpha \neq Cn(F_1)$, then that $(\otimes CN)$ implies $P_{\leq\alpha} \subseteq Cn(F_1)$.*

Therefore, only if the new information is accepted in the level higher than any other old information, $(\otimes CN)$ is satisfied.

Next we consider another extreme strategy of inserting new information: *the new information is accepted in the lowest degree*.

Theorem 3. *Let $\otimes$ be a revision function based on a POP $\Sigma = (K, \mathcal{P}, <)$ with ordinal type η. Let $\otimes_{F_1}$ be the revision function based on the minimal change $\Sigma^{F_1,\alpha}$ of Σ. If $\alpha \geq \eta$, then*

($\otimes CB$) *If $F_2 \cup (K \otimes F_1)$ is inconsistent, then $(K \otimes F_1) \otimes_{F_1} F_2 = K \otimes F_2$.*

The necessary condition for $(\otimes CB)$ is presented in the following proposition.

Proposition 2. *If $F_1 \not\subseteq K$, then $(\otimes CB)$ implies $K \otimes F_1 \subseteq P^{F_1,\alpha}_{\leq\alpha}$.*

Therefore to satisfy $(\otimes CB)$, new information should be accepted in the lowest degree unless it has been included in the belief set.

5 Conclusion and Discussion

In this paper we have presented a model-theoretical analysis of several typical postulates for iterated belief revision in the setting of multiple belief revision. The model we use in the paper is one of the typical ways to rank beliefs: assigning beliefs to ordinals. We have shown that Darwiche and Pearl's postulate $(\otimes C1)$ requires the change of orderings to be minimal. It doesn't put any restrictions on the way to rank new information. However, as the strengthening of $(\otimes C1)$, Boutilier's postulate $(\otimes CB)$ implies that new information should be accepted in the lowest degree of belief whereas Nayak *et al.*'s postulate $(\otimes CN)$ requires that new information should be accepted in highest level of firmness. These results provide an ontological base for the analysis of rationality of postulates for iterated belief revision.

As a side-product, we have proved that all the postulates discussed in the paper are consistent with Zhang and Foo's postulates for multiple revision. Therefore the multiple revision framework can be strengthened by choosing some of the postulates. Since the AGM postulates and Darwiche and Pearl's postulate $(\otimes C1)$ have received strong ontological support and are most intuitive. A framework that consists of these postulates would be most applicable. Zhang *et al.* used the framework in the construction of mutual revision functions and negotiation functions[23], which provides another ontology of iterated belief revision.

Proofs of Theorems

Proof of Lemma 1: i). For $(\otimes 7)$, $K\otimes(F_1\cup F_2) = (K\otimes F_1)\otimes_{F_1}(F_1\cup F_2) \subseteq (K\otimes F_1)+(F_1\cup F_2) = (K\otimes F_1)+F_2$. To show $(\otimes 8)$, assume that $F_2\cup(K\otimes F_1)$ is consistent. Then $(K\otimes F_1)+F_2 = (K\otimes F_1)+(F_1\cup F_2) = (K\otimes F_1)\otimes_{F_1}(F_1\cup F_2) = K\otimes(F_1\cup F_2)$.

ii) and ii) are straightforward. □.

Proof of Lemma 2: For the first part of the lemma, assume $A \in Cn(P_{\leq\alpha})$. Then there exist $A_1,\cdots,A_n \in P_{\leq\alpha}$ such that $A_1,\cdots,A_n \vdash A$. By Logical Constraint, $\max\{r(A_1),\cdots,r(A_n)\} \leq r(A)$. Therefore $A \in P_{\leq\alpha}$.

For the second part of the lemma, let $\mathcal{P} = \{P_\alpha : \alpha < \eta\}$, where $P_\alpha = \{A \in K : r(A) = \alpha\}$. It is easy to verify that $\mathcal{P}$ satisfy the logical constraint. □

Proof of Lemma 3: Suppose that $F \cup K$ is inconsistent. Then $\{r(A) : A \in K \text{ and } F \vdash \neg A\}$ is nonempty. Since $\mathcal{P}$ is well-ordered, there must be a minimum P_{min} in the set.

Now assume that $B \in K \otimes F$. According to Definition 2, there exists $A_0 \in K$ such that $F \vdash \neg A_0$ and

$$\forall C \in K((A_0 \vdash C \;\&\; F \vdash \neg C) \Rightarrow (r(C \vee B) < r(C) \text{ or } \vdash C \vee B)) \qquad (2)$$

Suppose that $r(A_1) = P_{min}$ and $F \vdash \neg A_1$. Let $A = A_0 \vee A_1$. Then $F \vdash \neg A$. By using Equation 2, we obtain $r(A \vee B) < r(A)$ or $\vdash A \vee B$, that is, $r(A \vee B) < P_{min}$ or $\vdash A \vee B$.

Conversely, assume that $A \in K$, $F \vdash \neg A$ and

$$b(A \vee B) < P_{min} \text{ or} \vdash A \vee B$$

If $\vdash A \vee B$, then obviously $B \in K \otimes F$. Otherwise, for any $C \in K$, if $F \vdash \neg C$ and $A \vdash C$, then $r(C) \geq P_{min}$. Thus $r(C \vee B) \leq r(A \vee B) < P_{min} \leq r(C)$. By Definition 1 we conclude that $B \in K \otimes F$. $\square$

Proof of Theorem 1: In the case that $(F_1 \cup F_2) \cup K$ or $(F_1 \cup F_2) \cup (K \otimes F_1)$ is consistent, the result is straightforward. Thus we assume that both $(F_1 \cup F_2) \cup K$ and $(F_1 \cup F_2) \cup (K \otimes F)$ are inconsistent. Let

$P_{min}^{F_1 \cup F_2} = \min\{r(A) : A \in K \ \& \ F_1 \cup F_2 \vdash \neg A\} = \beta$,

$P_{min\otimes}^{F_1 \cup F_2} = \min\{r^1(A) : A \in K \otimes F_1 \ \& \ F_1 \cup F_2 \vdash \neg A\} = \gamma$,

where r and r^1 is the rank with respect to Σ and $\Sigma^{F_1,\alpha}$, respectively. Then there exists $A_0 \in K$ such that $F_1 \cup F_2 \vdash \neg A_0$ and $r(A_0) = \beta$. Similarly, there exists $A_1 \in K \otimes F_1$ such that $F_1 \cup F_2 \vdash \neg A_1$ and $r^1(A_1) = \gamma$. Let $A = A_0 \vee A_1$. Then $F_1 \cup F_2 \vdash \neg A$, $A \in K$ and $A \in K \otimes F_1$. By the minimality of $P_{min}^{F_1 \cup F_2}$ and $P_{min\otimes}^{F_1 \cup F_2}$, we know that $r(A) = \beta$ and $r^1(A) = \gamma$.

Next we prove $\beta = \gamma$. On one hand, since $A \in K \cap (K \otimes F_1)$, by the construction of minimal change, $r^1(A) \leq r(A)$. Thus $\gamma \leq \beta$. On the other hand, $r^1(A) = \gamma$ implies that $A \in P_{\gamma}^{F_1,\alpha}$. Hence $P_{\leq\gamma} \cup F_1 \vdash A$. It follows that there exists a finite subset $\bar{F}_1$ of F_1 such that $P_{\leq\gamma} \vdash \neg(\wedge\bar{F}_1) \vee A$. Therefore $r(\neg(\wedge\bar{F}_1) \vee A) \leq \gamma$. Since $F_1 \cup F_2 \vdash (\wedge\bar{F}_1) \wedge \neg A$ and $\neg(\wedge\bar{F}_1) \vee A \in K$, by the minimality of β we have $r(\neg(\wedge\bar{F}_1) \vee A) \geq \beta$. Therefore we obtain that $\beta \leq r(\neg(\wedge\bar{F}_1) \vee A) \leq \gamma$, that is, $\beta \leq \gamma$.

Now we prove that $(K \otimes F_1) \otimes_{F_1} (F_1 \cup F_2) = K \otimes (F_1 \cup F_2)$. Assume that $B \in (K \otimes F_1) \otimes_{F_1} (F_1 \cup F_2)$. Then there exists $A_2 \in K \otimes F_1$ such that $F_1 \cup F_2 \vdash \neg A_2$ and

$$r^1(A_2 \vee B) < P_{min\otimes}^{F_1 \cup F_2} \text{ or} \vdash A_2 \vee B.$$

If $\vdash A_2 \vee B$, then $\neg A_2 \vdash B$. Therefore $B \in K \otimes (F_1 \cup F_2)$. If $r^1(A_2 \vee B) < P_{min\otimes}^{F_1 \cup F_2}$, let $r^1(A_2 \vee B) = \delta$. Then $P_{\leq\delta} \cup F_1 \vdash A_2 \vee B$. It follows that there exists a finite subset $\bar{F}_1$ of F_1 such that $P_{\leq\delta} \vdash \neg(\wedge\bar{F}_1) \vee A_2 \vee B$. Let $A_2' = \neg(\wedge\bar{F}_1) \vee A_2 \vee A$, where A was defined above. Therefore $r(A_2' \vee B) \leq \delta < P_{min\otimes}^{F_1 \cup F_2} = P_{min}^{F_1 \cup F_2}$. Since $A_2' \in K$ and $F_1 \cup F_2 \vdash \neg A_2'$, by Lemma 3, we yield that $B \in K \otimes (F_1 \cup F_2)$. We have proved $(K \otimes F_1) \otimes_{F_1} (F_1 \cup F_2) \subseteq K \otimes (F_1 \cup F_2)$.

For the other direction, assume that $B \in K \otimes (F_1 \cup F_2)$. Then there exists $A_3 \in K$ such that $F_1 \cup F_2 \vdash \neg A_3$ and

$$r(A_3 \vee B) < P_{min}^{F_1 \cup F_2} \text{ or} \vdash A_3 \vee B.$$

The case of $\vdash A_3 \vee B$ is obvious. In the case of $r(A_3 \vee B) < P_{min}^{F_1 \cup F_2}$, let $A_3' = A_3 \vee A$, where A was defined above. Then $A_3' \in K \otimes F_1$ and $F_1 \cup F_2 \vdash \neg A_3'$. It follows that

$$r^1(A_3' \vee B) \leq r(A_3' \vee B) \leq r(A_3 \vee B) < P_{min}^{F_1 \cup F_2} = P_{min\otimes}^{F_1 \cup F_2}$$

By Lemma 3, we yield that $B \in (K \otimes F_1) \otimes_{F_1} (F_1 \cup F_2)$. Therefore $K \otimes (F_1 \cup F_2) \subseteq (K \otimes F_1) \otimes_{F_1} (F_1 \cup F_2)$. $\square$

Proof of Theorem 2: For sufficiency, assume that $(F_1 \cup F_2) \cup K$ is consistent. Then $(K \otimes F_1) \otimes_{F_1} F_2 = (K + F_1) + F_2 = K + (F_1 \cup F_2) = K \otimes (F_1 \cup F_2)$. If $F_2 \cup (K \otimes F_1)$ is consistent, $(K \otimes F_1) \otimes_{F_1} F_2 = (K \otimes F_1) + F_2$. By $(\otimes 7)$ and $(\ominus 8)$, we know that the

result holds. Thus we can safely assume that both $(F_1 \cup F_2) \cup K$ and $F_2 \cup (K \otimes F_1)$ are inconsistent. Let

$P_{min}^{F_1 \cup F_2} = \min\{r(A) : F_1 \cup F_2 \vdash \neg A \text{ and } A \in K\} = \beta$

$P_{min\otimes}^{F_2} = \min\{r^1(A) : F_2 \vdash \neg A \text{ and } A \in K \otimes F_1\} = \gamma$

where r and r^1 is the rank over K and $K \otimes F_1$, respectively.

By $P_{min}^{F_1 \cup F_2} = \beta$, we know that there is $A_0 \in K$ such that $F_1 \cup F_2 \vdash \neg A_0$ and $r(A_0) = \beta$. Similarly, from $P_{min\otimes}^{F_2} = \gamma$, there is $A_1 \in K \otimes F_1$ such that $F_2 \vdash \neg A_1$ and $r^1(A_1) = \gamma$.

By $r^1(A_1) = \gamma$, we know that $P_{\leq\gamma} \cup F_1 \vdash A_1$. Then there exists a finite subset $\bar{F}_1$ of F_1 such that $\neg(\wedge\bar{F}_1) \vee A_1 \in K$(since $A_1 \in K \otimes F_1$) and $P_{\leq\gamma} \vdash \neg(\wedge\bar{F}_1) \vee A_1$. Thus $r(\neg(\wedge\bar{F}_1) \vee A_1) \leq \gamma$. On the other hand, $F_1 \cup F_2 \vdash (\wedge\bar{F}_1) \wedge \neg A_1$. By the minimality of β, $r(\neg(\wedge\bar{F}_1) \vee A_1) \geq \beta$. Therefore we have $\gamma \leq \beta$.

We are now ready to prove that $(K \otimes F_1) \otimes_{F_1} F_2 \subseteq K \otimes (F_1 \cup F_2)$. Suppose that $B \in (K \otimes F_1) \otimes_{F_1} F_2$. Then there exists $A' \in K \otimes F_1$ such that $F_2 \vdash \neg A'$ and

$$r^1(A' \vee B) < P_{min\otimes}^{F_2} \text{ or } \vdash A' \vee B \tag{3}$$

If $\vdash A' \vee B$, $F_2 \vdash B$. Thus $B \in K \otimes (F_1 \cup F_2)$. If $\nvdash A' \vee B$, let $r^1(A' \vee B) = \delta$. Then $A' \vee B \in P_{\delta}^{F_1,0}$. It follows that $P_{\leq\delta} \cup F_1 \vdash A' \vee B$. Since $(F_1 \cup F_2) \cup K$ is inconsistent, there exists a finite subset $\bar{F}_1$ of F_1 and finite subset $\bar{F}_2$ such that $\neg(\wedge\bar{F}_1) \vee \neg(\wedge\bar{F}_2) \in K$ and $P_{\leq\delta} \vdash \neg(\wedge\bar{F}_1) \vee A' \vee B$. Hence $r(\neg(\wedge\bar{F}_1) \vee \neg(\wedge\bar{F}_2) \vee A' \vee B) \leq \delta$. Since $\delta < P_{min\otimes}^{\bar{F}_2} = \gamma \leq \beta = P_{min}^{F_1 \cup F_2}$, we yield $r(\neg(\wedge\bar{F}_1) \vee \neg(\wedge\bar{F}_2) \vee A' \vee B) < P_{min}^{F_1 \cup F_2}$. By Lemma 3 we obtain that $B \in K \otimes (F_1 \cup F_2)$.

To prove that $K \otimes (F_1 \cup F_2) \subseteq (K \otimes F_1) \otimes_{F_1} F_2$, assume that $B \in K \otimes (F_1 \cup F_2)$. Then there exists $A'' \in K$ such that $F_1 \cup F_2 \vdash \neg A''$ and

$$r(A'' \vee B) < P_{min}^{F_1 \cup F_2} \text{ or } \vdash A'' \vee B \tag{4}$$

If $\vdash A'' \vee B$, then $F_1 \cup F_2 \vdash B$. It follows that there exists a finite subset $\bar{F}_2$ of F_2 such that $\neg(\wedge\bar{F}_2) \in K \otimes F_1$ and $F_1 \vdash \neg(\wedge\bar{F}_2) \vee B$. By the assumed special construction of minimal change ($\alpha = 0$), we have $r^1(\neg(\wedge\bar{F}_2) \vee B) = 0$. Since $F_1 \cup F_2$ is consistent and $P_0 \subseteq Cn(F_1)$, $P_{min\otimes}^{F_2} > 0$. Therefore $r^1(\neg(\wedge\bar{F}_2) \vee B) < P_{min\otimes}^{F_2}$. By Lemma 3 we have $B \in K \otimes F_1 \otimes_{F_1} F_2$.

In the case of $r(A'' \vee B) < P_{min}^{F_1 \cup F_2}$. Since $F_1 \cup F_2 \vdash \neg A''$, there exists a finite subset $\bar{F}_2'$ of F_2 such that $\neg(\wedge\bar{F}_2') \in K \otimes F_1$ and $F_1 \vdash \neg(\wedge\bar{F}_2') \vee A''$. Then we have $r^1(\neg(\wedge\bar{F}_2') \vee A'') = 0$. It follows that $r^1(\neg(\wedge\bar{F}_2') \vee A'' \vee B) = 0$. By the same argument above, we conclude that $B \in K \otimes F_1 \otimes_{F_1} F_2$. □

Proof of Proposition 1: Assume that $A \in P_{\leq\alpha}$ and $A \notin Cn(F_1)$. Suppose that $P_{\leq\alpha} \not\subseteq Cn(F_1)$. Then there exists $B \in Cn(F_1)$ such that $B \notin P_{\leq\alpha}$. Let $F_2 = \{\neg A \vee \neg B\}$. Since $P_{\leq\alpha} \cup F_1$ is consistent, we have $P_{min}^{F_1} = \min\{r(A) : A \in K \ \& \ F_1 \vdash \neg A\} > \alpha$. It follows that $r(A) \leq \alpha < P_{min}^{F_1}$. Thus $A \in K \otimes F_1$. On the other hand, since $B \in K \otimes (F_1 \cup F_2)$, $(\otimes CN)$ implies that $B \in (K \otimes F_1) \otimes_{F_1} F_2 = (K \otimes F_1) \otimes_{F_2} \{\neg A \vee \neg B\}$. If $\vdash (A \wedge B) \vee B$, we have $\vdash B$, which contradicts $B \notin P_{\leq\alpha}$. Therefore $r^1((A \wedge B) \vee B) < r^1(A \wedge B)$, that is, $r^1(B) < r^1(A \wedge B)$, or $r^1(B) < r^1(A)$, which also contradicts $B \notin P_{\leq\alpha}$. □

Proof of Theorem 3: Let $P^{F_2}_{min\otimes} = \min\{r^1(A) : F_2 \vdash \neg A \text{ and } A \in K \otimes F_1\} = \gamma$. Then there exists a sentence $A_0 \in K \otimes F_1$ such that $F_2 \vdash \neg A_0$ and $r^1(A_0) = \gamma$.

First we assume that $F_2 \cup K$ is inconsistent. Let $P^{F_2}_{min} = \min\{r(A) : F_2 \vdash \neg A \text{ and } A \in K\} = \beta$. Then there exists a sentence $A_1 \in K$ such that $F_2 \vdash \neg A_1$ and $r(A_1) = \beta$. Let $A = A_0 \vee A_1$. Then $A \in K \cap (K \otimes F_1) = K \ominus F_1$. It is not difficult to show that $\beta = r(A_1) = r(A) = r^1(A) = r^1(A_0) = \gamma$. Therefore $P^{F_2}_{min} = P^{F_2}_{min\otimes}$.

Assume that $B \in K \otimes F_2$. Then there exists $A_2 \in K$ such that $F_2 \vdash \neg A_2$ and

$$r(A_2 \vee B) < P^{F_2}_{min} \text{ or } \vdash A_2 \vee B$$

If $\vdash A \vee B$, then $B \in (K \otimes F_1) \otimes_{F_1} F_2$. Otherwise, $r(A_2 \vee B) < P^{F_2}_{min}$. Let $A' = A_0 \vee A_2$. It follows that $A' \in K \ominus F_1$ and $F_2 \vdash \neg A'$. By the construction of the minimal change, $r^1(A' \vee B) = r(A' \vee B) \leq r(A_2 \vee B) < P^{F_2}_{min} = P^{F_2}_{min\otimes}$. Thus $B \in (K \otimes F_1) \otimes_{F_1} F_2$. So $K \otimes F_1 \subseteq (K \otimes F_1) \otimes_{F_1} F_2$.

Conversely, Assume that $B \in (K \otimes F_1) \otimes_{F_1} F_2$. Then there exists $A_3 \in K \otimes F_1$ such that $F_2 \vdash \neg A_3$ and

$$r^1(A_3 \vee B) < P^{F_2}_{min\otimes} \text{ or } \vdash A_3 \vee B$$

The case of $\vdash A \vee B$ is trivial. So we only consider the other case. Let $A'' = A_1 \vee A_3$. By using the similar argument above, we can obtain $B \in K \otimes F_2$. Therefore $(K \otimes F_1) \otimes_{F_1} F_2 = K \otimes F_2$.

Now let's consider the case that $F_2 \cup K$ is consistent. Given $B \in K \otimes F_2 = K + F_2$, there exists a finite subset $\bar{F}_2$ of F_2 such that $\neg(\wedge\bar{F}_2) \in K \otimes F_2$ and $\neg(\wedge\bar{F}_2) \vee B \in K$. Thus $\neg(\wedge\bar{F}_2) \vee B \in K \ominus F_1$, which implies $r^1(\neg(\wedge\bar{F}_2) \vee B) < \eta \leq \alpha = P^{F_2}_{min\otimes}$. Thus $B \in (K \ominus F_1) \otimes_{F_1} F_2$. Conversely, if $B \in (K \ominus F_1) \otimes_{F_1} F_2$, there exists $A_4 \in K \otimes F_1$ such that $F_2 \vdash \neg A_4$ and

$$r^1(A_4 \vee B) < P^{F_2}_{min\otimes} \text{ or } \vdash A_4 \vee B$$

Then $A_4 \vee B \in K \ominus F_1 \subseteq K$. Thus $B \in K + F_2 = K \otimes F_2$. □

Proof of Proposition 2: Assume that $K \otimes F_1 \not\subseteq P^{F_1,\alpha}_{\leq\alpha}$. Then there exists $A \in K \otimes F_1$ such that $A \notin P^{F_1,\alpha}_{\leq\alpha}$. If $A \notin K$, then there exists a finite subset $\bar{F}_1$ of F_1 such that $\neg(\wedge\bar{F}_1) \vee A \in K$. If $r(\neg(\wedge\bar{F}_1) \vee A) \in P^{F_1,\alpha}_{\leq\alpha}$, then $A \in P^{F_1,\alpha}_{\leq\alpha}$, a contradiction. Therefore we can safely assume that $A \in K$. Since $F_1 \not\subseteq K$, there exists $B \in F_1$ such that $B \notin K$. Thus $A \in K + \{\neg A \vee \neg B\} = K \otimes \{\neg A \vee \neg B\}$. However, it is not hard to verify that $A \notin (K \otimes F_1) \otimes_{F_1} \{\neg A \vee \neg B\}$. This contradicts $(\otimes CB)$. □

References

1. C. E. Alchourrón, P. Gärdenfors and D. Makinson, On the logic of theory change: partial meet contraction and revision functions, *The Journal of Symbolic Logic* 50(2), 510–530, 1985.
2. C. Boutilier, Revision sequences and nested conditionals, in *Proc. 13th Int. Joint Conf. on Artificial Intelligence (IJCAI'93)*, 519–525, 1993.

3. C. Boutilier,Iterated revision and minimal change of conditional beliefs, *Journal of Philosophical Logic*, 1996(25), 262–305.
4. A. Darwiche and J. Pearl, On the Logic of Iterated Belief Revision, Proceedings of the fifth Conference on Theoretical Aspects of Reasoning about Knowledge, 5–23, 1994.
5. A. Darwiche and J. Pearl, On the logic of iterated belief revision, *Artificial Intelligence*, 89:1–29, 1997.
6. N. Friedman and J. Y. Halpern, Belief revision: A critique. In *KR'96,* 421–431, 1996.
7. P. Gärdenfors, *Knowledge in Flux: Modeling the Dynamics of Epistemic States,* (The MIT Press), 1988.
8. P. Gärdenfors, Belief revision: an introduction, in: P. Gärdenfors ed., *Belief Revision* (Cambridge University Press, Cambridge), 1–28, 1992.
9. S. Konieczny and R. P. Perez, A framework for iterated revision, *Journal of Applied Non-Classical Logics* 10(3–4), 339–367, 2000.
10. D. Lehmann, Belief revision, revised, In *Proc. 14th Int. Joint Conf. on Artificial Intelligence* (IJCAI'95), 1534–1540, 1995.
11. W. Li , A logical framework for evolution of specification. in *Programming Language and Systems*, (ESOP'94), LNCS 788, Springer-Verlag, 394–408, 1994.
12. S Lindström, A semantic approach to nonmonotonic reasoning: inference operations and choice, *Uppsala Prints and Preprints in Philosophy*, 1994, No. 10.
13. A. Nayak, Iterated belief change based on epistemic entrenchment, *Erkentnis* 41, 353–390,1994.
14. A. Nayak, N. Foo, M Pagnucco and A. Sattar, Changing conditional Beliefs unconditionally, *TARK-96*, 119–135, 1996.
15. P. Peppas, Well behaved and multiple belief revision, in: W. Wahlster edit, *Proceedings of 12th European Conference on Artificial Intelligence(ECAI-96)*, 90–94, 1996.
16. W. Spohn, Ordinal conditional functions: a dynamic theory of epestemic states, in W. L. Harper and B. Skyrms eds., *Causation in Decision, Belief Change, and Statistics*, Volume 2, 105–134, 1987.
17. M. A. Williams, Iterated theory base change: a computational model, In *Proc. 14th Int. Joint Conf. on Artificial Intelligence (IJCAI'95)*,1541–1547, 1995.
18. D. Zhang, Belief revision by sets of sentences, *Journal of Computer Science and Technology,* 1996, 11(2), 108–125.
19. D. Zhang, S. Chen, W. Zhu, Z. Chen, Representation theorems for multiple belief changes, in:*Proc. 15th Int. Joint Conf. on Artificial Intelligence (IJCAI'97)*, 89–94, 1997.
20. D. Zhang, S. Chen W. Zhu and H. Li, Nonmonotonic reasoning and multiple belief revision, in: M. Pollack edit, *Proceedings of the 15th International Joint Conference on Artificail Intelligence (IJCAI-97)*, Morgan Kaufman, 95–100.
21. D. Zhang and W. Li, Open logic based on total ordered partition model, *Science in China(Series E)*, 41(6), 1998.
22. D. Zhang and N. Foo, Infinitary belief revision, *Journal of Philosophical Logic*, 6(30), 525–574, 2001.
23. D. Zhang, N. Foo, T. Meyer and R. Kwok, Negotiation as mutual belief revision, in: *Proceedings of the 5th Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC'03)*, IJCAI-03, 2003.

System Description: DLV with Aggregates*

Tina Dell'Armi[1], Wolfgang Faber[2], Giuseppe Ielpa[1], Nicola Leone[1], Simona Perri[1], and Gerald Pfeifer[2]

[1] Department of Mathematics, University of Calabria
87030 Rende (CS), Italy
{dellarmi,ielpa,leone,perri}@unical.it
[2] Institut für Informationssysteme, TU Wien
1040 Wien, Austria
faber@kr.tuwien.ac.at
pfeifer@dbai.tuwien.ac.at

1 Introduction

DLV is an efficient Answer Set Programming (ASP) system implementing the (consistent) answer set semantics [GL91] with various language extensions like weak constraints [BLR00], queries, and built-in predicates. A strong point of DLV is the high expressiveness of its language, which is able to express very complex problems, even problems which are hard for the complexity class Δ_3^P. The language of DLV is therefore strictly more expressive than normal logic programming which cannot express any problem beyond NP. Another strong point of DLV is its robust and efficient implementation, which integrates algorithms and heuristics from the field of nonmonotonic reasoning with several optimization techniques from the field of deductive databases.

The expressiveness of the DLV language together with the robustness of the implementation make DLV well-suited for developing knowledge-based applications. Indeed, the DLV system is disseminated in academia, and presumably soon also in industry – the industrial exploitation of DLV in the emerging areas of Knowledge Management and Information Integration is the subject of two international projects funded by the European Commission, namely, INFOMIX (Boosting Information Integration, project IST-2002-33570) and ICONS (Intelligent Content Management System, project IST-2001-32429).

Despite this high expressiveness, encoding aggregations over a multiset of elements satisfying some conditions is often cumbersome and unintuitive, requiring ordering relations and recursive definitions using these orders. In this system description, we present an extension of DLV by aggregates #count, #sum, #times, #min, and #max, which greatly simplifies the encoding of those frequently occurring concepts. Moreover, in many cases also a computational benefit can be observed. The extension of the DLV language by aggregates has been proposed very recently [DFI+03a], and it has been further enhanced to also support unstratified aggregates [DFI+03b].

* The work was partially supported by the European Commission under projects IST-2002-33570 INFOMIX, IST-2001-32429 ICONS, and IST-2001-37004 WASP.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 326–330, 2004.

2 DLP$^{\mathcal{A}}$: Extending ASP by Aggregates

The language of DLV is disjunctive datalog under the consistent answer set semantics [GL91], commonly called *Answer Set Programming (ASP)*.

We assume familiarity with "classic" ASP and refer to [LPF$^+$02] for a detailed description of the full language of DLV. In what follows, we refer to atoms, literals, etc. without aggregates as *standard atoms*, *standard literals*, and so forth.

A (DLP$^{\mathcal{A}}$) *set* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{\, Vars : Conj \,\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of standard literals.[1] A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$, where $\bar{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a set, and f is a *function name* among $\#\texttt{count}$, $\#\texttt{min}$, $\#\texttt{max}$, $\#\texttt{sum}$, $\#\texttt{times}$.

Since symbolic or ground sets consist of tuples of constants, a projection on the first element of each tuple in the set is applied, yielding a multiset of constants on which the aggregate functions are effectively applied. Intuitively, $\#\texttt{count}$ evaluates to the number of elements, $\#\texttt{min}$ to the minimum element, and $\#\texttt{max}$ to the maximum element of the multiset. $\#\texttt{sum}$ evaluates to the sum of the numbers (0 in case of the empty set) and $\#\texttt{times}$ to the product of the numbers in the multiset (1 in case of the empty set). If the argument of an aggregate function does not belong to its domain, the aggregate evaluates to false and and DLV issues a warning.

An *aggregate atom* is $Lg \prec_1 f(S) \prec_2 Rg$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, and Lg and Rg (called *left guard*, and *right guard*, respectively) are terms. One of "$Lg \prec_1$" and "$\prec_2 Rg$" can be omitted; "$0 \leq$" and "$\leq +\infty$", respectively, are assumed then.

As an example, consider a predicate $person(Name, Age, Salary)$. The aggregate atom $19 < \#\texttt{min}\{A : person(N, A, S)\} \leq 30$ then evaluates to true iff the age of the youngest person is in the range $[20..30]$. $\#\texttt{sum}\{S : person(N, A, S), A > 30\} > 10000$, on the other hand, evaluates to true if the sum of the salaries of all persons exceeds 10000; it evaluates to false else.

Let $a_1, \ldots, a_n$ be classical literals (atoms possibly preceded by the classical negation symbol $-$) and $b_1, \ldots, b_m$ be classical literals or aggregate atoms and $n \geq 0, m \geq k \geq 0$. A *(disjunctive) rule* r is a formula

$$a_1 \ \texttt{v} \ \cdots \ \texttt{v} \ a_n \ \texttt{:-} \ b_1, \cdots, b_k, \ \texttt{not} \ b_{k+1}, \cdots, \ \texttt{not} \ b_m.$$

A *strong constraint* is a rule with empty head ($n = 0$). A *program* $\mathcal{P}$ is a finite set of rules and constraints.

The semantics of programs without aggregates is provided in [BLR00] as an extension of the classical answer set semantics given in [GL91]. The detailed semantics of the full language including aggregates can be found in [DFI$^+$03b].

[1] Intuitively, a symbolic set $\{X : a(X, Y), p(Y)\}$ stands for the set of X-values making $a(X, Y), p(Y)$ true. Note that $Conj$ may contain negative literals.

3 Representing Problems in DLP$^{\mathcal{A}}$

DLP$^{\mathcal{A}}$ is a superset of classic Answer Set Programming and thus allows to encode problems in a highly declarative fashion and solve problems of high computational complexity (Δ_3^P for the full language supported by DLV, for details we refer to [LPF$^+$02]).

In this section, we will focus on encodings employing the novel feature of aggregates, using the problem of Team Building, where a project team has to be built from a set of employees according to the following specifications:

p_1 The team consists of a certain number of employees.
p_2 At least a given number of different skills must be present in the team.
p_3 The sum of the salaries of the employees working in the team must not exceed the given budget.
p_4 The salary of each individual employee is within a specified limit.
p_5 The number of women working in the team has to reach at least a given number.

Suppose that our employees are provided by a number of facts of the form *emp(EmpId,Sex,Skill,Salary)*; the size of the team, the minimum number of different skills, the budget, the maximum salary, and the minimum number of women are specified by the facts *nEmp(N)*, *nSkill(N)*, *budget(B)*, *maxSal(M)*, and *women(W)*. We then encode each property p_i above by an aggregate atom A_i, and enforce it by an integrity constraint containing $\texttt{not}\, A_i$.

$$
\begin{aligned}
&in(I) \texttt{v} out(I) \texttt{:-}\ emp(I, Sx, Sk, Sa).\\
&\texttt{:-}\ nEmp(N), \texttt{not\#count}\{I : in(I)\} = N.\\
&\texttt{:-}\ nSkill(M), \texttt{not\#count}\{Sk : emp(I, Sx, Sk, Sa), in(I)\} \geq M.\\
&\texttt{:-}\ budget(B), \texttt{not\#sum}\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\} \leq B.\\
&\texttt{:-}\ maxSal(M), \texttt{not\#max}\{Sa : emp(I, Sx, Sk, Sa), in(I)\} \leq M.\\
&\texttt{:-}\ women(W), \texttt{not\#count}\{I : emp(I, f, Sk, Sa), in(I)\} \geq W.
\end{aligned}
$$

Intuitively, the disjunctive rule "guesses" whether an employee is included in the team or not, while the five constraints correspond one-to-one to the five requirements p_1-p_5. Thanks to the aggregates the translation of the specifications is surprisingly straightforward. The example highlights the usefulness of representing both sets and multisets in our language (recall that a multiset can be obtained by specifying more than one variable in $Vars$ of a symbolic set $\{ Vars : Conj \}$).

For instance, the encoding of p_2 requires a set, as we want to count *different* skills; two employees in the team having the same skill, should count once w.r.t. p_2. On the contrary, p_3 requires to sum the elements of a multiset; if two employees have the same salary, *both* salaries should be summed up for p_3. This is obtained by adding the variable I to $Vars$. The valuation of $\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\}$ yields the set $S = \{\langle Sa, I\rangle : Sa \textit{ is the salary of employee } I \textit{ in the team}\}$. Then, the sum function is applied on the multiset of the first components Sa of the tuples $\langle Sa, I\rangle$ in S.

4 Implementation and Usage

We have modified the implementation of DLV in order to deal with aggregates in the following way:

After parsing, each aggregate A is transformed such that both guards are present and both $\prec_1$ and $\prec_2$ are set to $\leq$. The conjunction $Conj$ of the symbolic set of A is replaced by a single, new atom Aux and a rule Aux `:-` $Conj$ is added to the program (the arguments of Aux being the distinct variables of $Conj$).

The instantiator, which transforms input with variables to ground programs in a bottom-up manner, ensures that all rules which define predicates appearing in an aggregate have been processed before it considers rules containing that aggregate. Furthermore, it identifies repeated occurrences of the same aggregate function in different rules and replaces them by a single internal representation. thus reducing the size of the output.

Finally, we designed an extension of the Deterministic Consequences operator of the DLV system [FLP99] to perform both forward and backward inferences on aggregate atoms, resulting in an effective pruning of the search space during model generation. We then extended the Dowling and Gallier algorithm [DG84] to compute a fixpoint of this operator in linear time using a multi-linked data structure of pointers.

A more detailed description of these aspects as well as some benchmarking can be found in [DFI$^+$03a].

While DLV with aggregates also offers a graphical user interface, the system per se is a command-line tool. It reads input from text files whose names are provided on the command-line, and can also obtain input from relation databases via an ODBC interface; any output is written to the standard output.

In its default mode, DLV computes all answer sets of its input. Command-line options like `-n=3` or `-filter=in` then can be used to request only a certain number of answer sets (3 in this case) or that only specific predicates (`in` in this case) are printed.

Detailed documentation, a full manual, and binaries of DLV with aggregates for various platforms are available at `http://www.dlvsystem.com`.

References

[BLR00] F. Buccafurri, N. Leone, and P. Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE TKDE*, 12(5), 2000.

[DFI$^+$03a] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *IJCAI 2003*, Acapulco, Mexico, August 2003. Morgan Kaufmann.

[DFI$^+$03b] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Semantics and Computation of Aggregate Functions in Disjunctive Logic Programming. Tech. Report INFSYS RR-1843-03-07, TU Wien, April 2003.

[DG84] W. F. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfability of Propositional Horn Formulae. *JLP*, 3:267–284, 1984.

[FLP99] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. *LPNMR'99*, pp. 177–191. Springer.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[LPF+02] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. Tech. Report cs.AI/0211004, arXiv.org, November 2002. Submitted to ACM TOCL.

GnT — A Solver for Disjunctive Logic Programs

Tomi Janhunen and Ilkka Niemelä

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O.Box 5400, FIN-02015 HUT, Finland
{Tomi.Janhunen,Ilkka.Niemela}@hut.fi

1 Introduction

Disjunctive logic programming under the stable model semantics [3] is a form of *answer set programming* (ASP) which is understood nowadays as a new logic programming paradigm. The basic idea is that a given problem is solved by devising a logic program such that the stable models of the program correspond to the solutions of the problem, which are then found by computing stable models for the program. The success of ASP is much due to efficient solvers, such as DLV [5] and SMODELS [9], which have been developed in recent years. Consequently, many interesting applications of the paradigm have emerged: planning, model checking, reachability analysis, and product configuration, just to mention some.

Of the two solvers mentioned above, DLV is more general, as it covers the whole class of disjunctive programs (DLPs) whereas only non-disjunctive programs can be handled with SMODELS. This raises the question whether the latter system can be extended to handle DLPs, too. This is non-trivial, as deciding whether a DLP has a stable model forms a Σ_2^p-complete decision problem whereas the problem remains NP-complete for non-disjunctive programs.

In [4], we have initiated an approach where stable models of DLPs are computed using SMODELS as a *core engine*. The approach is based on two program transformations using which the key tasks in computing disjunctive stable models can be reduced to computing stable models for normal programs. More specifically, the first transformation is needed in order to generate model candidates while the second implements a test for minimality — a central property of (disjunctive) stable models. We have implemented these transformations in an experimental solver GnT "Generate'n'Test" [7] designed for DLPs. The implementation is based on an architecture where two SMODELS search engines interact: one is responsible for generating model candidates for a DLP given as input and the other checks for the minimality of the candidates.

The rest of this system description is organized as follows. In Section 2, we give more details on the transformations discussed above and the method for computing disjunctive stable models using SMODELS as a core engine. Section 3 gives some examples how GnT and the related tools are used in practice. Section 4 presents some preliminary results from experiments with GnT and DLV.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 331–335, 2004.

2 Theoretical Background

In the sequel, we consider DLPs in the propositional case. Given a DLP P, we let $\mathrm{Hb}(P)$ stand for the set of propositional atoms that appear in P. An interpretation $M \subseteq \mathrm{Hb}(P)$ is a stable model of P iff M is a minimal model of the Gelfond-Lifschitz reduct P^M [3]. The first one of the program transformations given in [4] generates candidates that are potential stable models of P. Rather than generating all subsets of $\mathrm{Hb}(P)$, we try to produce as good candidates as possible. To this end, we distinguish two central properties of disjunctive stable models and impose them on the candidates. First of all, every stable model M of P satisfies all rules of P. This aspect is captured by the normal program $\mathrm{G1}(P)$, see (1) in Fig. 1. The basic idea is that if the body of a disjunctive rule $A \leftarrow B, {\sim}C$ is true in M, then $M \models a$ for some head atom $a \in A$. In fact, $\mathrm{G1}(P)$ is far more restrictive by design: $\mathrm{G1}(P)$ captures the stable models of P if P is disjunction-free (normal). The second key property is *supportedness* [1], i.e. each atom a true in a model $M \models P$ must have a supporting rule $A \leftarrow B, {\sim}C \in P$ such that $M \models B \cup {\sim}C$ and $M \not\models \bigvee(A - \{a\})$. The translation $\mathrm{Supp}(P)$, given as (2) in Fig. 1, encodes this property in terms of normal rules. The union $\mathrm{Gen}(P) = \mathrm{G1}(P) \cup \mathrm{Supp}(P)$ is a normal program whose stable models effectively capture a subset of the *supported models* of P [1].

The second program transformation from [4], given as (3) in Fig. 1, implements the required test for minimality. It is based on a technique presented in [6] and works as follows: given a DLP P a candidate model M is a minimal model of P^M iff the constructed normal program $\mathrm{Test}(P, M)$ has no stable model.

Hence, given a procedure for computing stable models for normal programs all stable models of a DLP P can be generated as follows: for each stable model M of $\mathrm{Gen}(P)$, decide whether $\mathrm{Test}(P, M)$ has a stable model and if this is not the case, output M as a stable model of P. This approach can be optimized by building model candidates gradually from the empty partial interpretation as the minimality test can be applied also to partial interpretations using the following result [4]. Given a DLP P and a (total) interpretation M if $\mathrm{Test}(P, M)$ has a

$$\begin{aligned}\mathrm{G1}(P) = {} & \{a \leftarrow {\sim}\hat{a}, B, {\sim}C \mid A \leftarrow B, {\sim}C \in P \text{ and } a \in A\} \cup \\ & \{\hat{a} \leftarrow {\sim}a \mid a \in \mathrm{Hb}(P)\} \cup \\ & \{f \leftarrow {\sim}f, {\sim}A, B, {\sim}C \mid A \leftarrow B, {\sim}C \in P\}. \end{aligned} \tag{1}$$

$$\begin{aligned}\mathrm{Supp}(P) = {} & \{a^{\mathrm{s}} \leftarrow {\sim}(A - \{a\}), B, {\sim}C \mid A \leftarrow B, {\sim}C \in P \text{ and } a \in A\} \cup \\ & \{f \leftarrow {\sim}f, a, {\sim}a^{\mathrm{s}} \mid a \in \mathrm{Hb}(P)\}. \end{aligned} \tag{2}$$

$$\begin{aligned}\mathrm{Test}(P, M) = {} & \{a \leftarrow {\sim}\hat{a}, B \mid A \leftarrow B \in P^M, a \in A \cap M, \text{ and } B \subseteq M\} \cup \\ & \{\hat{a} \leftarrow {\sim}a \mid a \in \mathrm{Hb}(P)\} \cup \\ & \{f \leftarrow {\sim}f, {\sim}A, B \mid A \leftarrow B \in P^M \text{ and } B \subseteq M\} \cup \\ & \{f \leftarrow {\sim}f, M\}. \end{aligned} \tag{3}$$

Fig. 1. Translations for generating model candidates and testing minimality [4]

stable model, then there is no (total) stable model M' of P such that $M \subseteq M'$. Hence, any partial interpretation M can be tested by treating it as a total one with undefined atoms taken to be false and if $\mathrm{Test}(P, M)$ has a stable model, all extensions of M can be pruned from the search space of model candidates.

The implementation [7] is based on the SMODELS system [9,8]. The basic idea behind the implementation is to create two instances of the SMODELS search engine, one that generates the model candidates gradually and one that tests if they are minimal. As the minimality test is computationally expensive, it is not applied to all partial interpretations but in the following way. Each time a (total) model candidate M of $\mathrm{Gen}(P)$ is found, it is checked if it is minimal. If it is not, then the procedure backtracks to the previous partial interpretation M' and performs the minimality test with $\mathrm{Test}(P, M')$ by viewing M' as a total model. This is done repeatedly until the test succeeds. Next, the search for further models is resumed. The implementation of the procedure discussed above consists of a few hundred lines of code [7] on top of the SMODELS system.

3 Examples

The source code of GNT is publicly available [7]. The program is to be compiled under the SMODELS (versions 2.*) source distribution, but precompiled Linux binaries are also available. GNT is to be used with the front-end LPARSE (preferably versions 1.0.13 and later), as it assumes to receive its input in the internal file format of the SMODELS system. The GNT system is used in practice as follows. In input files, disjunctive rules are written in the following syntax:

$$\texttt{a}_1\texttt{| ... |a}_k\texttt{ :- b}_1\texttt{, ... ,b}_l\texttt{, not c}_1\texttt{, ... , not c}_m\texttt{.}$$

The rules may include variables, but the domains of variables must be limited with domain predicates. The front-end LPARSE, which has to be invoked using the command line option `--dlp`, performs an instantiation for the rules so that its output is effectively a propositional disjunctive program in the internal format.

The stable models of a disjunctive logic program — stored in an input file `disjunctive.lp` — can be computed by executing a shell command line like "`lparse --dlp disjunctive.lp | gnt 0`". As with SMODELS, the command line argument `0` can be replaced by a positive integer n. Then the first n stable models are computed; given that so many stable models exist. The front-end LPARSE supports also other semantics proposed for disjunctive programs. Translations that enable the computation of *partial stable models* and *regular models* are produced by the command line options `--partial` and `-r`, respectively.

4 Experiments

In [4], we compare GNT with DLV. One of our test problems is the problem of finding a minimal model of a set of clauses in which some prespecified variables are true. This is a Σ_2^p-complete problem [2], which is easily transformed into a disjunctive logic program, as explained in [4]. Our test cases are based on random

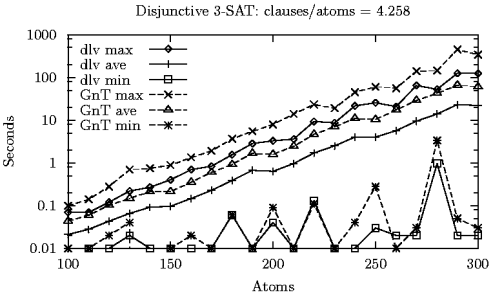


Fig. 2. Experimental Results: DLV vs. GNT

3-SAT problems generated using a fixed clauses-to-variables ratio $\frac{c}{v} = 4.258$ in the phase transition region. The number of variables v is increased by increments of 10 whereas the number of randomly chosen variables, which are supposed to be true in the minimal models being computed, is governed by $\lfloor 2v/100 \rfloor$. The results from this test are shown in Figure 2. The systems scale very similarly, but DLV seems to be roughly a constant factor faster than GNT. This is probably due to the overhead caused by the increased length of the normal programs that are needed for generating model candidates and checking minimality.

Acknowledgments. The author of the first version of GNT, also included as the file `example4.cc` in the smodels source distribution, is Patrik Simons. The second version, which is a derivative of the first one, was developed by Tomi Janhunen to speed up computation. We thank Tommi Syrjänen for implementing the support for disjunctive rules and alternative semantics in the front-end LPARSE. This research is partially funded by the Academy of Finland under the project "Applications of Rule-Based Constraint Programming" (#53695).

References

1. S. Brass and J. Dix. Characterizations of the (disjunctive) stable semantics by partial evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997.
2. T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15:289–323, 1995.
3. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
4. T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. CoRR: cs.AI/0303009, March 2003. Submitted for publication.
5. N. Leone et al. A disjunctive datalog system DLV (2002-04-12). Available in the World Wide Web: `http://www.dbai.tuwien.ac.at/proj/dlv/`, 2002.
6. I. Niemelä. A tableau calculus for minimal model reasoning. In *Proceedings of the Fifth Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 278–294, Terrasini, Italy, May 1996. Springer-Verlag.

7. P. Simons and T. Janhunen. GNT — a solver for disjunctive logic programs. Available in the World Wide Web: `http://www.tcs.hut.fi/Software/gnt/`, 2003.
8. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
9. P. Simons and T. Syrjänen. SMODELS and LPARSE – a solver and a grounder for normal logic programs. Available in the World Wide Web: `http://www.tcs.hut.fi/Software/smodels/`, 2002.

LPEQ and DLPEQ — Translators for Automated Equivalence Testing of Logic Programs*

Tomi Janhunen and Emilia Oikarinen

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science
P.O.Box 5400, FIN-02015 HUT, Finland
{Tomi.Janhunen,Emilia.Oikarinen}@hut.fi

1 Introduction

Answer set programming (ASP) is a logic programming paradigm where a problem at hand is solved in a declarative fashion by (i) writing a logic program whose *answer sets* correspond to the solutions of the problem; and by (ii) computing the answer sets of the program using a special purpose search engine. Despite the declarative nature of ASP, the development of logic programs resembles that in conventional programming languages. That is, a programmer often develops a series of improved programs for a particular problem, e.g., when optimizing the execution time and space. As a consequence, there is a need to ensure that subsequent programs, which differ in performance, yield the same output. In ASP, this leads us to the problem of checking whether given two logic programs P and Q give rise to the same answer sets. This notion corresponds to the *weak equivalence* of the programs P and Q, denoted by $P \equiv Q$ in the sequel.

A brute-force method to verify $P \equiv Q$ is to enumerate all answer sets of P and to check that each of them is also an answer set of Q, and vice versa. This approach is called NAIVE in [6,9]. Cross-checking answer sets in this way is time-consuming if the number of answer sets is high. To address this problem, we have developed translation-based methods for the automated verification of weak equivalence [6,9]. The rough idea is to translate programs P and Q into another logic program whose answer sets capture potential counter-examples to $P \equiv Q$. Then existing ASP implementations can be used to check the existence of counter-examples and special-purpose search engines need not be developed.

There are also other notions of equivalence that have been proposed for logic programs. Lifschitz el al. [7] consider P and Q to be *strongly equivalent*, denoted by $P \equiv_{\mathrm{s}} Q$, if and only if $P \cup R \equiv Q \cup R$ for all other programs R. Because of the arbitrary context R involved in the definition, checking $P \equiv_{\mathrm{s}} Q$ appears to be a more complicated task than checking $P \equiv Q$. Fortunately, the problem

* This research is affiliated with the project "Applications of Rule-Based Constraint Programming" (#53695) funded by the Academy of Finland.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 336–340, 2004.

of covering all possible contexts can be circumvented using a characterization of strong equivalence based on *SE-models* [11]. Inspired by this result, we have implemented a two-phased method for the automated verification of $P \equiv_{\mathrm{s}} Q$.

Our implementations, translators LPEQ [3] and DLPEQ [8], cover equivalence testing as follows: (i) weak equivalence of weight constraint programs [6] under a generalization of the stable model semantics [10]; (ii) classical equivalence of normal logic programs; (iii) strong equivalence [7,11] of normal logic programs under the stable model semantics [1]; and (iv) weak equivalence of disjunctive logic programs [9,8] supported by the GNT system [5] under the stable model semantics [2]. Many other syntactic extensions can be covered via translations and there are also further equivalence relations; see [9] for a summary.

We proceed as follows. Section 2 discusses some details of the translation-based method. In Section 3, we describe the current command line interfaces of LPEQ and DLPEQ and give some examples how the tools can be used for equivalence testing in practice. Section 4 concludes our experiences obtained from experiments with the two translator programs and the respective solvers.

2 Theoretical Background

The basic idea in the translation-based approach is to *translate* any two logic programs P and Q under consideration[1] into a single logic program $\mathrm{EQT}(P, Q)$ which has a stable model if and only if P has a stable model that is not a stable model of Q. Such stable models, if found, yield *counter-examples* to $P \equiv Q$. Consequently, the equivalence of P and Q can be established by showing that $\mathrm{EQT}(P, Q)$ and $\mathrm{EQT}(Q, P)$ have no stable models. The details of $\mathrm{EQT}(P, Q)$ can be found in [6] and the current implementation called LPEQ [3] covers the verification of weak equivalence for weight constraint programs [10].

We have also implemented other transformations that allow verifying the strong equivalence of normal logic programs. Turner's characterization of strong equivalence [11] implies that $P \equiv_{\mathrm{s}} Q$ if and only if P and Q are *classically equivalent*, denoted by $P \equiv_{\mathrm{c}} Q$, and for all interpretations M, the models $N \subseteq M$ of the Gelfond-Lifschitz reducts P_M and Q_M [1] coincide. The translator LPEQ includes the implementations of the respective translations $\mathrm{EQT}_{\mathrm{c}}(P, Q)$ and $\mathrm{EQT}_{\mathrm{s}}(P, Q)$ that are used analogously to $\mathrm{EQT}(P, Q)$, but for verifying $P \equiv_{\mathrm{s}} Q$. This has to be done phase-wise, i.e. the classical equivalence must be checked first. The reason is that the translation $\mathrm{EQT}_{\mathrm{s}}(P, Q)$ relies that $P \equiv_{\mathrm{c}} Q$ holds.[2]

Our translation-based approach is generalized to the disjunctive case in [9]. Counter-examples are defined as stable models M of P satisfying *either* (i) $M \not\models Q_M$; *or* (ii) $M \models Q_M$ and $M' \models Q_M$ for some $M' \subset M$. The respective translation $\mathrm{EQT}_{\mathrm{d}}(P, Q)$, which is used analogously to $\mathrm{EQT}(P, Q)$, captures both kinds of counter-examples at once. Yet another approach is based on two translations $\mathrm{EQT}_{\mathrm{d1}}(P, Q)$ and $\mathrm{EQT}_{\mathrm{d2}}(P, Q)$ which capture the two kinds of

[1] It is assumed that the Herbrand bases $\mathrm{Hb}(P)$ and $\mathrm{Hb}(Q)$ coincide.

[2] Note that $\mathrm{EQT}_{\mathrm{s}}(P, Q)$ need not be used if $\mathrm{EQT}_{\mathrm{c}}(P, Q)$ has a stable model.

```
lparse p1.lp > p1.sm
lparse p2.lp > p2.sm
lpeq p1.sm p2.sm | smodels 1
lpeq p2.sm p1.sm | smodels 1
```

```
lparse --dlp p1.lp > p1.sm
lparse --dlp p2.lp > p2.sm
dlpeq p1.sm p2.sm | gnt 1
dlpeq p2.sm p1.sm | gnt 1
```

Fig. 1. Examples on running LPEQ and DLPEQ

counter-examples separately. Then $\mathrm{EQT}_{\mathrm{d2}}(P, Q)$ is simplified given that counter-examples of the first kind do not exist. Thus $\mathrm{EQT}_{\mathrm{d1}}(P, Q)$ is to be used first.

3 Examples

The translators LPEQ [3] and DLPEQ [8] have been implemented in the C programming language under the Linux operating system. Both translators take two logic programs and command line options as their input and produce a translation for equivalence testing as their output. The input files are assumed to be in an internal format, as produced by the front-end LPARSE of the SMODELS system. Rules with variables can also be used, but LPARSE performs an instantiation for the rules. As an example, the weak equivalence of two *weight constraint programs*, once stored in the input files `p1.lp` and `p2.lp`, can be verified using LPEQ and SMODELS by giving the commands shown in Fig. 1 on the left-hand side. Classical and strong equivalence of *normal programs* can be verified using SMODELS in a similar fashion, but additional command line options `-c` and `-s`, respectively, have to be supplied for LPEQ. Furthermore, LPARSE (preferably version 1.0.13 or later) has to invoked using the command line option `-d all`. Otherwise LPARSE might apply program reductions that are sound for weak equivalence, but incorrect for classical/strong equivalence.

The other translator is used very similarly and it has been designed for disjunctive programs and the solver GNT [4]. In addition to disjunctive rules, DLPEQ supports *compute statements* which are extra constraints on stable models. In order to check the weak equivalence of two *disjunctive programs* `p1.lp` and `p2.lp`, the translator DLPEQ can be run using the commands listed in Fig. 1 on the right. Note that LPARSE has to be invoked with the command line option `--dlp` for disjunctive programs. By default, DLPEQ produces the one-phased translation $\mathrm{EQT}_{\mathrm{d}}(P, Q)$ as its output. Translations for the two-phased approach are produced by the command line options `-p [1|2]`.

A further command line option `-v` triggers a textual output for both translators, which we hope to be useful with solvers that are not compatible with the internal file format. In particular, to enhance interoperability with the DLV system, the output of DLPEQ can be adjusted by the command line option `--dlv`.

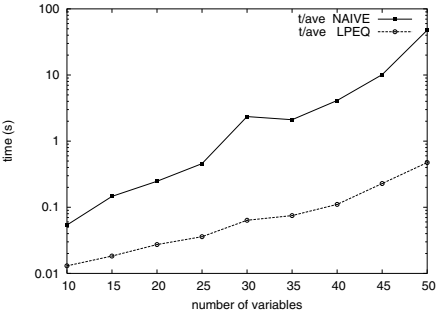


Fig. 2. Testing the equivalence of random 3-sat instances with clauses-to-variables ratio 4.

4 Experiments

We have carried out several experiments in order to compare the performance obtained with translations produced by LPEQ and DLPEQ with that of NAIVE. In Figure 2, we give the results for testing the weak equivalence of a normal logic program P, which corresponds to a random 3-sat instance, and a variant P' which is obtained from P by dropping one of its rules at random.

Our experiments suggest that the translation-based approach is superior to the naive one when the programs being tested possess many stable models, and in particular when the programs turn out to be not equivalent. But if the number of stable models is low or the programs under consideration do not possess stable models at all, then the naive cross-checking approach is likely to be faster. This is mainly because the translation $\mathrm{EQT}(P, Q)$ employs P as a subprogram to generate stable models of P. Furthermore, our preliminary experiments suggest that for normal programs, verifying strong equivalence is computationally more demanding than verifying weak equivalence. We believe that this is due to a greater search space for counter-examples, as the number of SE-models tends to be much higher than that of stable models.

References

1. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. MIT Press.
2. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
3. T. Janhunen. LPEQ 1.13 — a tool for testing the equivalence of logic programs. `http://www.tcs.hut.fi/Software/lpeq/`, 2002. Computer Program.
4. T. Janhunen. GnT (version 2) — a tool for computing disjunctive stable models. `http://www.tcs.hut.fi/Software/gnt/`, 2003. Computer Program.
5. T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. CoRR: cs.AI/0303009, March 2003.

6. T. Janhunen and E. Oikarinen. Testing the equivalence of logic programs under stable model semantics. In S. Flesca et al., editors, *Logics in Artificial Intelligence, Proceedings of the 8th European Conference*, pages 493–504, Cosenza, Italy, September 2002. Springer-Verlag. LNAI 2424.
7. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
8. E. Oikarinen. DLPEQ 1.9 — a tool for testing the equivalence of disjunctive logic programs. `http://www.tcs.hut.fi/Software/lpeq/`, 2003. Computer Program.
9. E. Oikarinen and T. Janhunen. Verifying the equivalence of logic programs in the disjunctive case. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, 2004. This volume.
10. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
11. H. Turner. Strong equivalence made easy: Nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4–5):609–622, 2003.

DLV^{DB}: Bridging the Gap between ASP Systems and DBMS*s*

Nicola Leone, Vincenzino Lio, and Giorgio Terracina

Dipartimento di Matematica, Università della Calabria,
Via Pietro Bucci, 87036 Rende (CS), Italy
{leone,lio,terracina}@mat.unical.it

Abstract. The exploitation of ASP systems for solving real application problems pointed out the need of combining the expressive power of ASP programming with the efficient data management features of existing DBMS*s*. This paper presents DLV^{DB}, an extension to the DLV system allowing to instantiate logic programs directly on databases and to handle input and output data distributed on several databases.

1 Introduction

In recent years, the development of ASP systems like DLV [2] and Smodels [5] has renewed the interest in the area of non-monotonic reasoning and declarative logic programming for solving real world problems in a number of application areas. However, many interesting problems are either "data intensive", such as the automatic correction of census data [4], or they produce huge ground programs that can not be handled in main-memory data structures; as a consequence, efficient and effective data management techniques are mandatory. Moreover, several application contexts provide input data as stored in (possibly distributed) databases; for instance, DLV is being used in a data integration framework[1] where input data are generated by wrappers and stored in distributed databases.

The considerations above put into evidence the urgent need of combining the expressive power of ASP with the efficient data management features of DBMS*s*. Indeed, ASP provides an expressive power that goes far beyond that of SQL3, whereas good DBMS*s* provide very efficient query optimization mechanisms.

DLV^{DB} provides a first, initial contribution in this setting, bridging the gap between ASP systems and DBMS*s*. It is an extension of the DLV system which handles the instantiation of input programs directly on database and the management of input and output data distributed on several databases. DLV^{DB} combines the experience in optimizing ASP programs gained within the DLV project with the well assessed data management capabilities of existing DBMS*s*.

The present version of DLV^{DB} supports a small subset of the DLV language; however, it provides a well established infrastructure for the interoperation with

[1] In the context of the Information Society Technologies programme of the European Commission, Future and Emerging Technologies INFOMIX project - IST-2001-33570.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 341–345, 2004.

databases and allows the application of several optimization techniques already developed or under development in the DLV project (such as magic sets [1]). Finally, the architecture of DLV^{DB} has been designed so as to allow easy extensions to the more powerful capabilities of the DLV language, such as disjunctions.

We plan to test DLV^{DB} with various DBMS*s*, among which some main-memory databases, in order to verify the performances of the system with different data management technologies.

The present implementation of DLV^{DB} allows to handle normal stratified programs; in particular both true negation and negation as failure are supported. Moreover, all built-in predicates allowed in DLV are also supported in DLV^{DB}. Program safety is a necessary condition for a correct instantiation process; finally, identification of hard constraints violations is also supported even if, at present, the system just warns the user about the existence of these violations.

Since DLV^{DB} has been designed mainly for applications on (distributed) databases, it allows the user to specify some auxiliary directives for handling the interaction between DLV^{DB} and a set of databases. In particular, the user can specify the working database, in which the system has to perform the instantiation, and a set of table definitions; note that each table must be mapped into one of the program predicates. Facts can reside on separate databases and they can be obtained as views on different tables. Finally, the user can choose to copy the entire output of the instantiation or parts thereof in different databases. It is worth pointing out that these directives provide DLV^{DB} with high flexibility in source data location and output management.

2 Architecture

In this section we describe the architecture of DLV^{DB}. It has been developed following the general philosophy of reusing as much as possible existing modules of the DLV system. This allowed us to take advantage of optimization techniques, such as program rewritings [3], previously developed for DLV. When running DLV^{DB}, the user can choose either to exploit the database instantiation features of DLV^{DB} or to run the standard main-memory DLV version.

Figure 1 illustrates the architecture of the instantiator module of DLV^{DB}. In the figure, the boxes marked with DLV are the ones already developed in the DLV system. An input program $\mathcal{P}$ is first analyzed from the parser which encodes the rules in the intensional database (IDB) in a suitable way and builds an extensional database (EDB) in main-memory data structures from the facts specified in the program. If the DLV^{DB} instantiation is chosen, facts can be stored in input databases and the parser produces no EDB in main-memory. Then, a rewriting procedure, optimizes the rules in order to get an equivalent program $\mathcal{P}'$ that can be instantiated more efficiently and that can lead to a smaller ground program. The dependency graph builder computes the dependency graph of $\mathcal{P}'$, its connected components and a topological ordering of these components. Finally, if the DLV^{DB} instantiation is selected, the DB Instantiator module is activated; otherwise instatiation is handled by the standard DLV

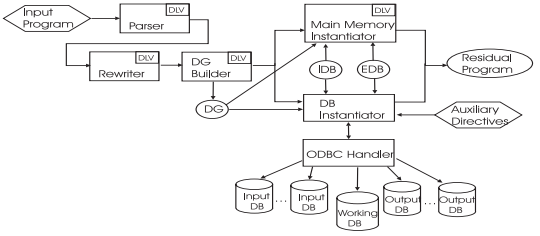


Fig. 1. Architecture of the instantiator module of DLV^{DB}.

main memory instantiator. In both cases, the output is the grounding of the input program.

The DB Instantiator module receives: *(i)* the IDB and the EDB (if not empty) generated by the parser, *(ii)* the Dependency Graph (DG) generated by the dependency graph builder and *(iii)* some auxiliary directives specifying the interaction between DLV^{DB} and the databases. Since the input program is supposed to be normal and stratified, the DB Instantiator evaluates completely the program and no further module is employed after the grounding.

All the instantiation steps are performed directly on the working database through the execution of SQL statements and no data is loaded in main-memory in any phase of the process. This allows DLV^{DB} to be completely independent from the dimension of both the input data and the produced ground program.

Input data (the facts) can reside in several input databases accessible through the internet; however, in order to improve efficiency, they are copied into the local working database. Finally, the results (or parts thereof) of the computation can be transferred to some output databases.

Communication with databases is performed via ODBC. This allows both to be independent from a particular DBMS and to handle distributed databases.

2.1 Implementation Issues

The DB Instantiator first maps all the predicates into the corresponding database tables, as specified by the auxiliary directives; default mappings are applied when needed. Then it stores the facts present in the EDB into the corresponding database tables. Note that DLV^{DB} allows some or all the facts to be already stored in the database; if this happens, they are not loaded in the main-memory EDB by the parser.

After all the mappings and input data have been prepared, the instantiation starts. Presently, DLV^{DB} adopts the simple Naive algorithm [6] for the instantiation. In the first phase of the implementation, we have chosen a simple instantiation algorithm to concentrate our attention on both the translation of DLV rules into SQL statements and the management of the interoperation between DLV^{DB} and the involved databases. However, the extension to more efficient semi-naive approaches will be quite straightforward and implemented soon.

The instantiation of each rule consists in the execution of the SQL statement representing it. One of the main objectives in the implementation of DLV^{DB} has been that of associating one single SQL statement with each rule of the program, without the support of main-memory data structures for the instantiation. This allows DLV^{DB} to fully exploit optimization mechanisms implemented in the DBMS*s* and to minimize the "out of memory" problems caused by limited main-memory dimensions.

The main guidelines for the translations are the following: *(i)* variable bindings determine join operations among tables; *(ii)* the SELECT part of the SQL statement is determined by the variables in the head of the rule and the associated bindings, *(iii)* the FROM part of the statement is determined by the predicates composing the body of the rule; *(iv)* variable bindings, negation and built-in predicates are handled in the WHERE part of the statement; *(v)* the INSERT INTO section is determined by the predicate in the head of the rule.

We show next an example of translation from a DLV rule to an SQL statement. Consider the rule $p(X, W) : -a(X, Y, Z), not\ b(Z), c(Y, W)$; it is instantiated by the SQL statement[2]:

```
INSERT INTO p ( SELECT a.att1, c.att2 FROM a,c WHERE a.att2=c.att1 AND
                                    (a.att3) NOT IN (SELECT b.att1 FROM b))
```

Acknowledgment. This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-33570 INFOMIX project and the IST-2001-37004 WASP project.

References

1. C. Cumbo, W. Faber, G. Greco, and N. Leone. Enhancing the Magic-Set Method for Disjunctive Datalog Programs. Technical report.
2. Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, and Gerald Pfeifer. System Description: DLV. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 409–412. Springer Verlag, September 2001.
3. Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.
4. E. Franconi, A. Laureti Palma, N. Leone, S. Perri, and F. Scarcello. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In *In* Proc. of LPAR 2001, pages 561–578, 2001.

[2] Here we use the notation t.att$_i$ to indicate the i-th attribute of the table t. Actual attribute names can be determined from the auxiliary directives.

5. Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A System for Answer Set Programming. In Chitta Baral and Mirosław Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.
6. J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

Cmodels-2: SAT-Based Answer Set Solver Enhanced to Non-tight Programs

Yuliya Lierler[1] and Marco Maratea[2]

[1] Department of Computer Sciences, University of Texas, Austin, USA
yuliya@cs.utexas.edu
[2] Dipartimento di Informatica, Sistemistica e Telematica, Università di Genova, Genova, Italy
marco@mrg.dist.unige.it

1 Introduction

Answer set programming is a new programming paradigm proposed in [1] and [2], and based on the answer set semantics of Prolog [3]. It is well known that an answer set for a logic program is also a model of the program's completion [4]. The converse is true when the logic program is "tight" [6,5]. Lin and Zhao [7] showed that for non-tight programs the models of completion which do not correspond to answer sets can be eliminated by adding to the completion what they called "loop formulas". Nevertheless, their solver ASSAT[1] has some disadvantages: it can work only with basic rules, and it can compute only one answer set. Answer set solver CMODELS-1[2] [12] is a system that computes answer sets for logic programs that are tight or can be transformed into tight programs, and does not suffer from these limitations.

We are going to present a new system CMODELS-2[1], that is able to fix ASSAT's disadvantages. Another attractive feature of the new system is that it organizes the search process more efficiently than ASSAT, because it does not explore the same part of the search tree more than once. In the rest of the paper we omit number 2 in the name of the system.

2 General Information

The input language of CMODELS is generated by the preprocessor LPARSE. The input may contain negation as failure, cardinality expressions ("constraint literals" in the terminology of [9, Section 5.3]) and choice rules [9, Section 5.4]—two constructs widely used in answer set programming.[3] These constructs are eliminated in favor of nested expressions in the sense of [10] using the method described in [11]. CMODELS is an answer set solver that uses SAT solvers as search engines. It may invoke different SAT solvers for finding solutions, namely SIMO [4], MCHAFF and ZCHAFF [5], and RELSAT [6]. The system can be easily

[1] http://assat.cs.ust.hk/
[2] http://www.cs.utexas.edu/users/tag/cmodels/
[3] The input can also contain general weight expressions ("weight literals"). However, optimize statements [9, Section 5.6] are not allowed.
[4] http://www.mrg.dist.unige.it/~sim/simo/
[5] http://www.ee.princeton.edu/~chaff/
[6] http://www.satlib.org/Solvers/SAT/REL_SAT.2/

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 346–350, 2004.

extended to new SAT solvers. This allows us to take advantage of the rapid progress in the area of satisfiability solvers.

The algorithm of CMODELS for tight programs is identical to the one in CMODELS-1 and is described in details in [12]. The difference appears only when the program is non-tight. In such case CMODELS needs to verify that each model of completion is indeed the answer set. Whenever the model of completion does not correspond to any answer set the computation of loop formulas is performed. We will talk about the mechanism of dealing with non-tight programs in the following section.

3 CMODELS on Non-tight Programs

In case of non-tight programs the algorithm of CMODELS is similar to the one of system ASSAT described in [7], but there are several differences.

First, CMODELS works with more general programs, and uses extended definition of a loop formula [8]. The technique for computing loop formulas is nevertheless very similar to the one described in [7]. Second, CMODELS introduces a "generate and test" approach in accordance to the SAT-based methodology [13] with such SAT solvers as SIMO and ZCHAFF. According to the SAT-based methodology, we first:

- *generate* a total propositional model satisfying the propositional clauses, and then
- *test* if the generated valuation is an answer set.

Both ASSAT and CMODELS use a "generate and test" approach, but when CMODELS invokes SIMO or ZCHAFF for the computation:

- the SAT solver never explores the same part of the search tree (this is only partially true in case of ZCHAFF communication [7]).
- it is possible to compute more than one answer set.

The main difference between ASSAT and CMODELS is that ASSAT uses SAT solvers as black-boxes and CMODELS, in case of invoking SIMO and ZCHAFF , does not. Details will be presented in a future paper.

4 CMODELS Options

CMODELS command line is:

```
   cmodels number [-mc] [-zc] [-si] [-sia] [-rs] [rs1] [-t] [-nt]
[-s] [-le] [-bj]
```

`number` Specifies the number of answer sets to be found. 0 stands for "compute all answer sets"; 1 is the default.
`-mc` MCHAFF is used for finding answer sets (default).
`-zc` ZCHAFF is used for finding answer sets.
`-si` SIMO is used for finding answer sets.
`-sia` SIMO is used for finding answer sets with an ASSAT-like algorithm.
`-rs` RELSAT version 2 is used for finding answer sets.

[7] CMODELS uses an incremental learning option of ZCHAFF.

Instance name	CMODELS SIMO (-le)	SMODELS	CMODELS SIMO assat alg.
np40c	(42) 1.56	2.49	(27) 16.52
np60c	(106) 8.80	21.45	(35) 76.12
np70c	(217) 26.46	42.86	(41) 139.50
np80c	(223) 37.87	79.78	(44) 241.55
np100c	(286) 78.93	200.43	(51) 561.94
np120c	(698) 314.93	430.98	mem
np150c	(1074) 841.91	1171.38	mem
4xp20.1	(2) 73.26	timeout	(1) 2.03
4xp20.3	(13) 82.90	0.01	(5) 1.56

Fig. 1. Complete and hand-coded graphs CMODELS employing learning vs. SMODELS vs. CMODELS employing assat algorithm.

`-rs1` RELSAT version 1.1.2 is used for finding answer sets.
`-t` Stands for omitting the tightness check [12]; CMODELS expects input program to be tight. By default, the tightness check is performed.
`-nt` Stands for omitting the tightness check [12]; CMODELS expects input program to be non-tight and invokes the procedures related to non-tight programs.
`-s` Stands for omitting the simplification step [12]. By default the simplification is performed.
`-le` and `-bj` are relevant to the invocation of SIMO. They refer to "learning" and "backjumping" using SAT terminology; by default [-le] is performed. For more details see [13].

5 Performance of CMODELS

In this section we report some experimental data. All experiments were run on two identical Pentium IV 1.8GHz processors with 512MB of RAM, DDR 266MHz, running Linux.

In Figures 1 and 2, CPU time is reported in seconds as the sum of *user* and *system* time using *time* UNIX command. *timeout* means that the process was stopped after one hour, *mem* means that the process reached the memory limit of the machine. The number in parentheses is the number of times CMODELS checked that a propositional model does not correspond to an answer set. We compare CMODELS only with SMODELS: this is due to the fact that DLV is optimized for disjunctive logic programs and ASSAT works with programs of more limited syntax. Nevertheless, we have implemented an ASSAT-like algorithm in our framework. We focus on finding one answer set.

First we evaluate the performances of CMODELS on Hamiltonian circuits (HC) benchmarks, in particular on some publicly available complete graphs and on hand-coded graphs. [8] We also built some bigger complete graphs. Complete graphs are in particular interesting because, as observed in [7], they have exponential number of loops. In Fig. 1 we present the comparison of CMODELS using SIMO with "learning", SMODELS and CMODELS using SIMO employing the algorithm of ASSAT. For complete graphs, we can see a clear edge between CMODELS running SIMO with "learning" and as black-box communication. This seem to point out the usefulness of a "non black-box" communication with

[8] `http://assat.cs.ust.hk/assat-1.0.html`

Instance Name	CMODELS MCHAFF	CMODELS SIMO (-le)	SMODELS
dp_6.formula1-i-O2-b8	(28) 6.17	(6) 0.43	0.46
dp_8.formula1-i-O2-b10	(24) 28.72	(41) 6.61	5.08
dp_8.formula1-s-O2-b8	(15) 5.14	(14) 0.94	1.83
dp_10.formula1-i-O2-b12	(21) 51.36	(162) 14.34	428.85
dp_10.formula1-s-O2-b9	(24) 19.47	(36) 3.51	29.05
dp_12.formula1-i-O2-b14	(29) 469.83	(14) 81.80	timeout
dp_12.formula1-s-O2-b10	(69) 96.95	(93) 7.56	949.95

Fig. 2. Non-tight Bounded Model Checking CMODELS using MCHAFF employing ASSAT-like algorithm and SIMO employing learning vs. SMODELS

a SAT solver. The advantage in comparison with SMODELS is smaller: around a factor of two. Instead, the results on hand-coded graphs are quite negative.

The second domain we demonstrate in Fig. 2 is bounded LTL model checking problem[9]. In this domain, we can see a clear edge between CMODELS using SIMO both versus CMODELS using MCHAFF as a black-box and SMODELS.

Acknowledgments. We are grateful to Enrico Giunchiglia and Vladimir Lifschitz for their comments related to the subject of this paper, and to Armando Tacchella for his help related to the integration of the systems. This work is partially supported by MIUR and ASI.

References

1. V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, 1999.
2. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.
3. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.
4. K. Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
5. Yu. Babovich, E. Erdem, and V. Lifschitz. Fages' theorem and answer set programming.[10] In *Proc. Eighth Int'l Workshop on Non-Monotonic Reasoning*, 2000.
6. F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
7. F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAAI-02*, 2002.
8. J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs.[11] In *Proc. ICLP-03*, To appear.

[9] `http://www.tcs.hut.fi/~kepa/experiments/boundsmodels/`

[10] `http://arxiv.org/abs/cs.ai/0003042.`

[11] `http://www.cs.utexas.edu/users/appsmurf/papers/disjunctive.ps.`

9. T. Syrjanen. Lparse manual.[12] 2003.
10. V. Lifschitz, L. R. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.
11. P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. Theory and Practice of Logic Programming, to appear.
12. Yu. Lierler and V. Lifschitz. Computing answer sets using program completion[13] Unpublished draft.
13. A. Armando, C. Castellini, E. Giunchiglia, F. Giunchiglia and A. Tacchella. SAT-Based Decision Procedures for Automated Reasoning: a Unifying Perspective. In Festschrift in Honor of Jörg H. Siekmann, to appear in LNAI, Springer-Verlag 2002.

[12] `http://www.tcs.hut.fi/software/smodels/lparse.ps.gz.`
[13] `http://www.cs.utexas.edu/users/yuliya/cmodels.ps.`

WSAT(CC) — A Fast Local-Search ASP Solver

Lengning Liu and Mirosław Truszczyński

Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA

Abstract. We describe *WSAT(CC)*, a local-search solver for computing models of theories in the language of propositional logic extended by cardinality atoms. *WSAT(CC)* is a processing back-end for the logic PS^+, a recently proposed formalism for answer-set programming.

1 Introduction

WSAT(CC) is a local-search solver for computing models of theories in the logic PL^{cc}, the propositional logic extended by cardinality atoms [3,4]. It can serve as a processing back-end for the logic PS^+ [3], an *answer-set programming* (ASP) formalism based on the language of predicate calculus and, hence, different from typical ASP systems that have origins in logic programming.

A *clause* in the logic PL^{cc} is a formula $\alpha_1 \wedge \ldots \wedge \alpha_r \rightarrow \alpha_{r+1} \vee \ldots \vee \alpha_s$, where each α_i, $1 \leq i \leq s$, is a propositional atom, or a *cardinality atom* (*c-atom*, for short) — an expression $k\{a_1, \ldots, a_n\}m$, where a_i are propositional atoms and k, m and n are integers such that $0 \leq k \leq m \leq n$. A set of atoms M is a *model* of a c-atom $k\{a_1, \ldots, a_n\}m$ if $k \leq |M \cap \{a_1, \ldots, a_n\}| \leq m$. With this definition, the semantics of clauses and theories in the logic PL^{cc} is a straightforward extension of the semantics of propositional logic. PL^{cc} theories arise by grounding theories in the ASP logic PS^+ by means of the grounder program *asppsgrnd* [3].

We discuss here an implementation of *WSAT(CC)*. We restrict the discussion to most essential concepts and options only. For more details and bibliography on related work on propositional logic extended by c-atoms and pseudo-boolean constraints, we refer to [4], which introduced *WSAT(CC)*, and to [2].

2 *WSAT(CC)* — A Brief Description and a List of Options

As other WSAT-like local-search solvers [6,7], *WSAT(CC)* searches for models in a series of *tries*, starting with a random assignment of truth values to atoms. Each try consists of steps, called *flips*, which produce "new" truth assignments by *flipping* the truth values of some of the atoms. If a flip produces a satisfying assignment, this try is terminated and another one starts. *WSAT(CC)* supports several strategies to select atoms for flipping. All of them require a parameter called the *noise level*. It determines the probability of applying a random walk step in order to escape from a local minimum. The maximum numbers of tries

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 351–355, 2004.

and flips, and the noise level are set from the command line by means of the options `-t`, `-c` and `-N`, respectively.

WSAT(CC) is different from other similar algorithms in the way in which it computes the *break-count* of an atom (used to decide which atom to flip) and in the way it executes a flip. The choice of the break-count computation method or of the way a flip is defined determines a particular local-search strategy in *WSAT(CC)*. At present, *WSAT(CC)* supports three basic methods.

Virtual break-count. We define virtual break-counts with respect to a propositional theory, in which c-atoms are replaced by their equivalent propositional representations. However, in the actual computation we use the original theory (with c-atoms) rather than its propositional-logic counterpart (with c-atoms removed), as the latter is usually exponentially larger. To invoke virtual break-count method, we use the option `-VB`. The virtual break-count method is applicable with all PL^{cc} theories and is a default method of *WSAT(CC)*.

Double flip. It applies only to *simple* PL^{cc} theories that are specified by the following two conditions: (a) all the c-atoms appear in unit clauses, and (b) all the sets of atoms in the c-atoms are pairwise disjoint. A flip is designed so that all unit clauses built of c-atoms remain satisfied. Thus, on occasion, two atoms will change their truth values in one flip step. The break-count is defined with respect to regular propositional clauses as in *WSAT*. To invoke this method, we use the option `-DF`.

Permutation flip. It applies to theories, in which c-atoms are used solely to specify permutations (for instance, when defining an assignment of queens in the n-queens problem). Flips realize an inverse operation on permutations and, hence, transform a permutation into another permutation. As a consequence, all unit clauses built of c-atoms are always satisfied. To accomplish that, four atoms must have their truth values changed in one flip step. The break-count is defined with respect to regular propositional clauses of the theory in the same way as in *WSAT*. We invoke this method with the option `-PF`.

3 *WSAT(CC)* — Input, Output, and How to Invoke It

WSAT(CC) accepts input files containing PL^{cc} theories described in a format patterned after that of CNF DIMACS. The first line is of the form `p <na> <nc>`, where `na` and `nc` are the number of propositional atoms and clauses in the theory, respectively. The following lines list clauses. A clause $\alpha_1 \wedge \ldots \wedge \alpha_r \rightarrow \alpha_{r+1} \vee \ldots \vee \alpha_s$, is written as `A1 ... Ar , A(r+1) ... As`, where each `Ai` is a positive integer (representing the corresponding atom α_i), or an expression of the form {`k m C1 ... Cn`} (representing a c-atom $k\{a_1, \ldots, a_n\}m$).

WSAT(CC) outputs models that it finds as well as several statistics to standard output device (or, depending on the options used, to a file in a user-readable format). It also creates a file `wsatcc.stat` that stores records summarizing every call to *WSAT(CC)* and key statistics pertaining to the computation.

Typical call to *WSAT(CC)* looks as follows: `wsatcc -f file -t 200 -c 150000 -N 10 100`. It results in *WSAT(CC)* looking for models to the PL^{cc}

theory specified in `file`, by running 200 tries, each consisting of 150000 flips. The noise level is set at 10/100 (=0.1).

4 *WSAT(CC)* Package

WSAT(CC) solver and several related utilities can be obtained from `http://www.cs.uky.edu/ai/wsatcc/`. *WSAT(CC)* works on most Unix-like operating systems that provide `gcc` compiler. The utilities require Perl 5 or greater. For more details on installation, we refer to [2].

5 Performance

Our experiments demonstrate that *WSAT(CC)* is an effective tool to compute models of *satisfiable* PL^{cc} theories and can be used as a processing back-end for the ASP logic PS^{+}. In [4], we showed that *WSAT(CC)* is often much faster than a local-search SAT solver *WSAT* and has, in general, a higher success rate (likelihood that it will find a model if an input theory has one). In [1], we used *WSAT(CC)* to compute several new lower bounds for van der Waerden numbers. Here, we will discuss our recent comparisons of *WSAT(CC)* with *WSAT(OIP)* [7], a solver for propositional theories extended with pseudo-boolean constraints (for which we developed utilities allowing it to accept PL^{cc} theories).

We tested these programs on PL^{cc} theories encoding instances of the vertex-cover and *open n-queens* problems[1]. We generated these theories by grounding appropriate PS^{+} theories extended with randomly generated problem instances.

Table 1 shows results obtained by running *WSAT(CC)* (both *-VB* and *-DF* versions are applicable in this case) and *WSAT(OIP)* to find vertex covers of sizes 1035, 1040 and 1045 in graphs with 2000 vertices and 4000 edges. The first column shows the size of the desired vertex cover and the number of graphs (out of 50 that we generated), for which we were able to find a solution by means of at least one of the methods used. The remaining columns summarize the performance of the three algorithms used: *WSAT(CC)-VB*, *WSAT(CC)-DF*, and *WSAT(OIP)*. The entries show the *time*, in seconds, needed to complete computation for all 50 instances and the *success rate* (the percentage of cases where the method finds a solution to all the instances, for which at least one method found a solution).

The results show that *WSAT(CC)-VB* is faster than *WSAT(CC)-DF*, which in turn is faster that *WSAT(OIP)*. However, *WSAT(CC)-VB* has generally the lowest success rate while *WSAT(CC)-DF*, the highest.

We note that we attempted to compare *WSAT(CC)* with *smodels* [5], a leading ASP system. We found that for the large instances that we experimented with *smodels* failed to terminate within the time limit that we allocated per

[1] In the open n-queens problem, given an initial "attack-free" assignment of k ($k < n$) queens on the $n \times n$ board, the goal is to assign the remaining $n - k$ queens so that the resulting assignment is also "attack-free".

Table 1. Vertex cover: Large Graphs

Family	*WSAT(CC)-VB*	*WSAT(CC)-DF*	*WSAT(OIP)*
1035 (9 / 50)	1453/77%	3426/100%	9748/11%
1040 (24 / 50)	1166/95%	2464/100%	7551/100%
1045 (36 / 50)	991/86%	1610/100%	6365/100%

instance. That is not surprising, as the search space is prohibitively large for a complete method and *smodels* is a complete solver.

The open n-queens problem allowed us to experiment with the method *-PF* (permutation flip). It proved extremely effective. We tested it for the case of 50 queens with 10 of them preassigned. We generated 100 random preassignments of 10 queens to a 50×50 board and found that 55 of them are satisfiable. We tested the four algorithms only on those satisfiable instances. The results are shown in Table 2.

Table 2. Open n-Queens: $N = 50$, 10 preassigned

Family	*WSAT(CC)-VB*	*WSAT(CC)-PF*	*WSAT(OIP)*	*smodels*
50+10(55 / 55)	20/1539/100%	9/768/100%	76/1459/100%	908/10%

Here, we include another measurement for local search solvers. The second number shows the average number of flips each method uses in finding one solution. *WSAT(CC)-VB* is faster than *WSAT(OIP)* even though they have the similar number of flips. *WSAT(CC)-PF* is even more powerful because it uses the fewest number of flip and is the fastest. Smodels can only find solutions for 6 instances within the 1000-second limit and turns out to be the slowest.

We tested the version *-PF* with one of the encodings of the Hamiltonian-cycle problem and discovered it is much less effective there. Conditions under which the version *-PF* is effective remain to be studied.

Acknowledgments. This research was supported by the National Science Foundation under Grants No. 0097278 and 0325063.

References

1. M.R. Dransfield, V.M. Marek, and M. Truszczyński. Satisfiability and computing van der Waerden numbers. In *Proceedings of SAT-2003*. LNAI, Springer Verlag, 2003.
2. D. East, L. Liu, S. Logsdon, V. Marek, and M. Truszczyński. ASPPS user's manual, 2003. `http://www.cs.uky.edu/aspps/users_manual.ps`.
3. D. East and M. Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of KI-2001*, LNAI 2174. Springer Verlag, 2001. Full version submitted for publication (available at `http://xxx.lanl.gov/abs/cs.LO/0211033`).

4. L. Liu and M. Truszczyński. Local-search techniques in propositional logic extended with cardinality atoms. In *Proceedings of CP-2003*. LNCS, Springer Verlag, 2003.
5. 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.
6. B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of AAAI-94*. AAAI Press, 1994.
7. J.P. Walser. Solving linear pseudo-boolean constraints with local search. In *Proceedings of AAAI-97*. AAAI Press, 1997.

Smodels with CLP—A Treatment of Aggregates in ASP

Enrico Pontelli, Tran Cao Son, and Islam Elkabani

Department of Computer Science
New Mexico State University
{epontell,tson,ielkaban}@cs.nmsu.edu

1 Introduction

Many practical systems have been recently proposed to support execution of *Answer Set Programming (ASP)* [13,7,1,10]. Our objective is to introduce different types of *aggregates* in ASP. Database languages (e.g., SQL) use aggregate functions—e.g., *sum*, *count*, and *max*—to obtain summary information from a database. Aggregates have been shown to significantly improve compactness and clarity of programs in various flavors of logic programming [9,6]. We expect to gain similar advantages from the introduction of aggregations in ASP.

Example 1. Consider a job scheduling problem, where we have a total number of machines (`resource(Max)`) and a number of jobs. Each job uses one machine and lasts for a certain duration (`duration(Job,Time)`). We can define a predicate $active(Job, Time)$ and use the aggregate $count$ to calculate the number of active jobs at time $Time$. The number of active jobs cannot be greater than the number of machines:

```
active(Job,Time) :- time(Time), duration(Job, Len), start(Job,Init), Time >= Init, Time < Init+Len.
                 :- time(T), resource(Max), count(T, active(Job,T)) > Max.
```

Example 2. ([11]) Let $owns(X, Y, N)$ denote the fact that company X owns a fraction N of the shares of the company Y. We say that a company X *controls* a company Y if the sum of the shares it owns in Y together with the sum of the shares owned in Y by companies controlled by X is greater than half of the total shares of Y.[1]

```
control(X, X, Y, N) :- owns(X,Y,N).                control(X, Z, Y, N) :- control(X,Z), owns(Z,Y,N).
fraction(X,Y,N) :- sum(M,control(X,Z,Y,M)) = N.    control(X,Y)        :- fraction(X,Y,N), N >0.5.
```

A significant body of research has been developed in the database and in the constraint programming communities exploring the theoretical foundations and, in a more limited fashion, the algorithmic properties of aggregation constructs in logic programming (e.g. [9,15,11,5]). More limited attention has been devoted to the more practical aspects related to computing in logic programming in presence of aggregates. In [2], it has been shown that aggregates can be encoded in ASP (e.g., example 1 above). The main disadvantage of this proposal is that the encoding contains many intermediate variables, thus making the grounding phase expensive. Recently, a number of proposals to extend logic programming with aggregate functions have been developed, including work on the use of aggregates in the ASET system [8], extensive work on sets grouping [14,6], and the excellent implementation of aggregates in *dlv* [4]. The approach proposed in this

[1] For the sake of simplicity we omitted the domain predicates required by *smodels*.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 356–360, 2004.

work accomplishes the same objectives as [4,8]. The novelty lies in the technique used to support aggregates. Following the spirit of our previous efforts [6,3], we integrate different *constraint solving* technologies to support sets and aggregates. In this paper, we describe an inference engine, obtained by integrating *smodels* with a finite-domain solver, capable of executing *smodels* program with aggregates. The engine is meant to be used in conjunction with front-ends capable of performing high-level constraint handling of sets and aggregates (as in [6]).

2 Integrating a Constraint Solver in an Answer Set Solver

We now describe the most relevant aspects of our system, that will be referred as *smodels–ag* hereafter. The general idea of our solution is to employ *finite domain constraints* to encode the aggregates in a program. Each atom appearing in an aggregate is represented as a variable with domain $0..1$; the whole aggregate is expressed as a constraint involving such variables. E.g., given the atoms $p(1), p(2), p(3)$, the aggregate $sum(A, p(A))<3$ will lead to the constraint

$$X[1]::0..1,\ X[2]::0..1,\ X[3]::0..1,\ X[1]*1+X[2]*2+X[3]*3\ \#< 3$$

where $X[1], X[2], X[3]$ are constraint variables corresponding to $p(1), p(2), p(3)$ respectively.

Syntax: The input language accepted by the *smodels-ag* system is analogous to the language of *smodels*, with the exception of a new class of literals—the *aggregate* literals. Aggregate literals are of the form $F(X, \bar{Y}, Goal[X, \bar{Y}])$ **Op** $Result$, where

- F is the aggregate function—currently we accept the functions sum, $count$, min, max;
- X is the grouped variable while $\bar{Y}$ are the existentially quantified variables;
- $Goal[X, \bar{Y}]$ is an atom ($\bar{Y}$ is empty) or an expression of the type $atom_1[X, \bar{Y}]$: $atom_2[\bar{Y}]$;
- **Op** is one of the relational operators drawn from the set $\{=, !=, <, >, <=, >=\}$;
- $Result$ is either a variable or a numeric constant.

The variables $X, \bar{Y}$ are locally quantified within the aggregate. At this time, the aggregate literal cannot play the role of a domain predicate—thus other variables appearing in an aggregate literal (e.g., $Result$) are treated in the same way as variables appearing in a negative literal.

In *smodels-ag*, we have opted for relaxing the stratification requirement present in [4,8], which avoids the presence of recursion through aggregates. The price to pay is the possibility of generating non-minimal models [6,9]; on the other hand, the literature has highlighted situations where stratification of aggregates prevents natural solutions to problems [11,5].

System Architecture: The overall structure of *smodels-ag* is shown in Fig. 1. The current implementation is built using *smodels (2.27)* and the ECLiPSe (5.4) constraint solver. At this stage it is a prototype aimed at investigating the feasibility of the proposed ideas. `Preprocessing`. The Preprocessing module is composed of three sequential steps. In the *first* step, a program – called *Pre-Analyzer* – is used to perform a number of simple syntactic transformations of the input program. The transformations are mostly aimed at rewriting the aggregate literals in a format acceptable by *lparse*. The *second* step

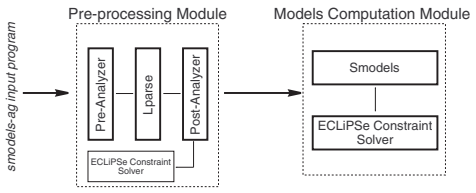


Fig. 1. Overall System Structure

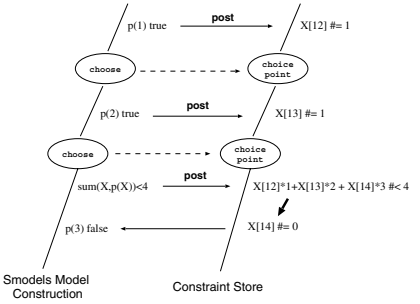


Fig. 2. Communication *smodels* to *ECLiPSe*

executes the *lparse* program on the output of the pre-analyzer, producing a grounded version of the program encoded in the format required by *smodels* (i.e., with a separate representation of rules and atoms). The *third* step is performed by the *Post-Analyzer* program whose major activities are:

- Identification of the dependencies between aggregate literals and atoms contributing to such aggregates; these dependencies are explicitly included in the output file. (The *lparse* output format is extended with a fourth section, describing these dependencies.)
- Generation of the constraint formulae encoding the aggregate; e.g., an entry like *"57 sum(x,use(8,x),3,greater)"* in the atom table (describing the aggregate *sum(X,use(8,X)) > 3*) is converted to *"57 sum(3,[16,32,48],"X16 * 2 + X32 * 1 + X48 * 4 + 0 #> 3")"* ($16, 32, 48$ are indices of the atoms *use(8,_)*).
- Simplification of the constraints making use of the truth values discovered by *lparse*.

`Models Computation`. The Model Computation module (Fig. 1) is in charge of generating the models from the input program. The module consists of a modified version of *smodels* interacting with an external finite domain constraint solver (in this case, the ECLiPSe solver).

As in *smodels*, each atom in the program has a separate internal representation—including aggregate literals. In particular, each aggregate literal representation maintains information regarding what program rules it appears in. The representation of each aggregate literal is similar to that of a standard atom, with the exception of some additional fields; these are used to store an ECLiPSe structure representing the constraint associated to the aggregate. Each standard atom includes a list of pointers to all the aggregate literals depending on such atom.

The main flow of execution is directed by *smodels*. In parallel with the construction of the model, our system builds a *constraint store* within ECLiPSe. The constraint store maintains *one conjunction* of constraints, representing the level of aggregate instantiation achieved so far. Each time a non-aggregate atom is made true or false, a new constraint is posted in the constraint store. If i is the index of such atom within *smodels*, and the atom is made true (false), then the constraint $X[i]\#{=}1$ ($X[i]\#{=}0$) is posted. (Fig. 2, first two `post` operations).

The structure of the computation developed by *smodels* is reflected in the structure of the constraints store (see Fig. 2). In particular, each time *smodels* generates a choice point (e.g., as effect of guessing the truth value of an atom), a corresponding choice point has to be generated in the store. Similarly, whenever *smodels* detects a conflict and initiates backtracking, a failure has to be triggered in the store as well. Observe that choice points and failures can be easily generated in the store using the *repeat* and *fail* predicates of ECLiPSe.

Since aggregate literals are treated by *smodels* as standard program atoms, they can be made true, false, or guessed. The only difference is that, whenever their truth value is decided, a different type of constraint will be posted to the store—i.e., the constraint that encodes the aggregate (Fig. 2, third posting). If the aggregate literal is made false, then a negated constraint will be posted (negated constraints are obtained by applying the $\#\backslash+$ ECLiPSe operator).

The *expand* procedure of *smodels* requires minor changes. An aggregate literal may become true/false not only as the result of the declarative closure computation, but also because enough evidence has been accumulated to prove its status. E.g., if the truth value of all atoms involved in the aggregate has been established, then the aggregate can be immediately evaluated. Observe that the constraints posted to the store have an active role during the execution:

- constraints can provide feedback to *smodels* by forcing truth or falsity of previously uncovered atoms (truth value is unknown at that time). E.g., if the constraint $X[12]{*}1 + X[13]{*}2 + X[14]{*}3\#{<}4$ is posted to the store (corresponding to the aggregate $sum(X, p(X)){<}10$) and $X[12]\#{=}1$ and $X[13]\#{=}1$ have been previously posted (e.g., $p(1)$ and $p(2)$ are true), then it will force $X[3]\#{=}0$, i.e., $p(3)$ to be false (Fig. 2, last step).
- inconsistencies in the constraint store have to be propagated to the *smodels* computation.

3 Discussion and Conclusions

The prototype implementing these ideas has been completed and used on a pool of benchmarks. Performance is acceptable, but we expect to obtain significant improvements by refining the interface with ECLiPSe. Combining a constraint solver with *smodels* brings many advantages:

- since we are relying on an external constraints solver to effectively handle the aggregates, the only step required to add new aggregates (e.g., $times$, avg) is the generation of the appropriate constraint formula during preprocessing;
- the constraint solvers are very flexible; e.g., by making use of Constraint Handling Rules we can implement different strategies to handle constraints and new constraint operators;
- the constraint solvers automatically perform some of the optimizations described in [4];
- it is straightforward to allow the user to declare aggregate instances as *eager*; in this case, instead of posting only the corresponding constraint to the store, we will also post a *labeling*, forcing the immediate resolution of the constraint store (i.e., guess the

possible combinations of truth values of selected atoms involved in the aggregate). In this way, the aggregate will act as a generator of solutions instead.

We believe this approach has advantages over previous proposals. The use of a general constraint solver allows us to easily understand and customize the way aggregates are handled (e.g., allow the user to select eager vs. non-eager treatment); it also allows us to easily extend the system to include new form of aggregates, by simply adding new type of constraints. Furthermore, the current approach relaxes some of the syntactic restriction imposed in other proposals (e.g., stratification of aggregations). The implementation requires minimal modification to the *smodels* system and introduces insignificant overheads for regular programs. The prototype confirmed the feasibility of this approach. Future work includes:

- further relaxation of some of the syntactic restrictions. E.g., the use of labeling allows the aggregates to "force" solutions, so that the aggregate can act as a generator of values; this may remove the need to include domain predicates to cover the result of the aggregate (e.g., the *safety* condition used in *dlv*).
- development of an independent grounding front-end; the current use of a pre-analyzer is dictated by the limitations of *lparse* in dealing with syntactic extensions.

References

1. Y. Babovich and V. Lifschitz. Computing Answer Sets Using Program Completion.
2. C. Baral. *Knowledge Representation, reasoning, and declarative problem solving*, Cambridge, 2003.
3. A. Dal Palu' et al. Integrating Finite Domain Constraints and CLP with Sets. *PPDP*, ACM, 2003.
4. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, G. Pfeifer. . Aggregate Functions in Disjunctive Logic Programming. IJCAI, 2003.
5. M. Denecker et al. Ultimate well-founded and stable semantics for logic programs with aggregates. In *ICLP*, Springer. 2001.
6. A. Dovier, E. Pontelli, and G. Rossi. Intensional Sets in CLP. *ICLP*, Springer Verlag, 2003.
7. T. Eiter et al. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *KRR*, 1998.
8. M. Gelfond. Representing Knowledge in A-Prolog. *Logic Programming & Beyond*, Springer, 2002.
9. D. Kemp and P. Stuckey. Semantics of Logic Programs with Aggregates. In *ILPS*, MIT Press, 1991.
10. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of Logic Programs By SAT Solvers. *AAAI'02*.
11. K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *JCSS*, 54:79–97, 1997.
12. K. A. Ross et al. Foundations of aggregation constraints. *TCS*, 193(1–2), 1998.
13. P. Simons et al. Extending and Implementing the Stable Model Semantics. *AIJ*, 138(1–2), 2002.
14. O. Shmueli et al. Compilation of Set Terms in the Logic Data Language (LDL). *JLP*, 12(1/2), 1992.
15. A. Van Gelder. The Well-Founded Semantics of Aggregation. In *PODS*, ACM Press, 1992.

nlp: A Compiler for Nested Logic Programming*

Vladimir Sarsakov[1], Torsten Schaub[1**], Hans Tompits[2], and Stefan Woltran[2]

[1] Institut für Informatik, Universität Potsdam, Postfach 90 03 27, D–14439 Potsdam, Germany
{sarsakov,torsten}@cs.uni-potsdam.de
[2] Institut für Informationssysteme, TU Wien, Favoritenstraße 9–11, A–1040 Wien, Austria
{tompits,stefan}@kr.tuwien.ac.at

Abstract. nlp is a compiler for nested logic programming under answer set semantics. It is designed as a front-end translating nested logic programs into disjunctive ones, whose answer sets are then computable by disjunctive logic programming systems, like dlv or gnt. nlp offers different translations: One is polynomial but necessitates the introduction of new atoms, another is exponential in the worst case but avoids extending the language. We report experimental results, comparing the translations on several classes of benchmark problems.

1 Nested Logic Programs and Their Compilation

Nested logic programs allow for arbitrarily nested formulas in heads and bodies of logic program rules under answer set semantics [4]. Nested expressions can be formed using conjunction, disjunction, and the negation-as-failure operator in an unrestricted fashion. Previous results [4] 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 [3] and gnt [2]. 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 a polynomial translation among both formalisms. In [5], we provide such a polynomial translations of nested logic programs into disjunctive ones. This translation introduces new atoms reflecting the structure of the original program; in the sequel, we refer to it as the *structural translation*. Likewise, we refer to the possibly exponential one as being *language-preserving*.

2 The nlp System

We have implemented both translations as a front-end to dlv [3] and gnt [2]. Both systems represent state-of-the-art implementations for disjunctive logic programs under answer set semantics. The resulting compiler, called nlp, is implemented in Prolog and is publicly available at

* The first two authors were supported by the German Science Foundation (DFG) under grant SCHA 550/6, TP C. All authors acknowledge support within IST-2001-37004 project WASP.

** Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.

V. Lifschitz and I. Niemelä (Eds.): LPNMR 2004, LNAI 2923, pp. 361–364, 2004.

```
p;not_t:-not q,q.
p;not_t:-not q,not s.
not_t:-not t.
:-t,not_t.
```

```
p;not_t:-not q,l1.
q;not_s:-l1.
l1:-q.
l1:-not s.
not_t:-not t.
:-t,not_t.
not_s:-not s.
:-s,not_s.
```

Fig. 1. Resulting (optimized) code for program Π under the **(a)** language-preserving translation (`example.dlp`); and **(b)** structural translation (`example.htl`).

`http://www.cs.uni-potsdam.de/~torsten/nlp/`.
The current implementation runs under SICStus and SWI Prolog and comprises roughly 1000 lines of code. In addition, it contains a number of optional improvements that allow for more compact code generation.

Consider program $\Pi = \{p \leftarrow \neg(q \vee \neg t) \wedge (q \vee \neg s)\}$, where '$\neg$' expresses negation as failure; it is expressed by `p :- not (q;not t), (q;not s).` Once a file is read into nlp, it is subject to multiple transformations. Most of these transformations are rule-centered in the sense that they apply in turn to each rule. While the original file is supposed to have the extension `nlp`, the result of the respective compilation is indicated by the extension `dlp` (language-preserving) and `htl` (structural), respectively. The system is best used with the command `nlp2`T`2`S for $T \in \{$`dlp`, `htl`$\}$ and $S \in \{$`dlv`, `gnt`$\}$; e.g., `nlp2htl2gnt(ex)` applies the structural translation (htl) to file `ex.nlp` and calls `gnt` on the result. Applying both transformations to our example Π yields the files in Figure 1a and 1b, respectively. The first rule in Figure 1a illustrates that we (currently) refrain from any sophisticated logical simplifications. On the other hand, the implemented versions of our translations are optionally improvable (via a flag) in two simple yet effective ways: First, no labels are generated for literals and second only a single label is generated for n-ary disjunctions or conjunctions. All our experiments are conducted using this optimization. Also, three different schemes for label generation are supported; here, we adhere to a number-oriented version (as seen in Figure 1b by `l1`).

3 Experimental Analysis

In view of the already diverging sizes on our toy example, it is interesting to compare the resulting images of our translations in a systematic way. To this end, we have implemented two different benchmark series that produce scalable problem instances.

Normal form series. The first one generates nested logic programs whose rule heads and bodies consist of two-level disjunctive and/or conjunctive normal form formulas. The generation of such nested logic programs is guided by 10 parameters: on the one hand, the number of variables, rules, facts, and constraints; and, on the other hand, the structure of heads and bodies is fixed by three parameters: the number of connectives on the first and second level plus the number of literals connected to the first level. The placement of literals (p, $\neg p$, and $\neg\neg p$), respectively, is done randomly. These parameters are also reflected

in the name of the source file. E.g., file `test_50.20_5_0.3_2_0.2_3_1.30.2.nlp` comprises a program over an alphabet of 50 variables, containing twenty rules, such as

```
p42,p4;p11,not p50;p46,p26 :-
        (not not p10;p37;p27),(p20;p34;p39),not not p11.
```

plus five facts and no constraints, expressed by `20_5_0`. While the primary connective of the head is a disjunction, namely '`;`', the one of the body is conjunction '`,`'. Although this can be arbitrarily fixed, it is currently set this way in order to provoke a significant blow-up when translating programs by means of successive applications of distributivity. The above head parameters, '`3_2_0`', enforce heads (being disjunctions) composed of 3 conjunctions, containing themselves 2 literals, along with zero literal disjuncts. The body parameters, '`2_3_1`', lead to conjunctions with 2 disjunctions, containing 3 literals, and 1 literal conjunct, viz. '`not not p11`' in the rule above. Finally, the filename ends with an estimated blow-up factor for the language-preserving translation, viz. '`30.2`' in our example.

Given a nested logic program Π, its blow-up factor, $\theta(\Pi)$, is computed as follows:

$$\theta(\Pi) = \frac{\lceil r\cdot(u_1+u_3+v_1+v_3)\cdot v_2^{v_1}\cdot u_2^{u_1}\rceil+\lceil h\cdot(u_1+u_3)\cdot u_2^{u_1}\rceil+\lceil b\cdot(v_1+v_3)\cdot v_2^{v_1}\rceil}{[(r+h)\cdot(u_1\cdot u_2+u_3)]+[(r+b)\cdot(v_1\cdot v_2+v_3)]},$$

where r stands for the number of rules; h, b gives the number of heads and bodies in the program; u_1, u_2, u_3 are the head parameters; and v_1, v_2, v_3 are the body parameters. The denominator of $\theta(\Pi)$ gives the size of Π in terms of its number of literals, whilst the numerator captures the size of the disjunctive program resulting from the language-preserving translation.

Cardinality constraints series. The second generator is based on the idea that (single headed) logic programs with cardinality literals can be transformed into nested logic programs [1]. A cardinality literal has the form $m\{L_1, \ldots, L_n\}n$ and means informally that the resulting answer set must contain at least m and at most n literals out of $\{L_1, \ldots, L_n\}$; see [6] for a detailed semantics. Such literals can then be used in the same way as ordinary literals in the head as well as the body of rules. As above, we start with generating a parameterized program with cardinality literals. For brevity, we omit further details and refer the interested reader to the URL below.

Experiments. The above described generators, along with the fully detailed test series, can be downloaded at

`http://www.cs.uni-potsdam.de/~torsten/nlp/bench/`.

In all, we generated 4661 nested logic programs, all of which where translated by each of the translations and the respective image was subsequently passed to `dlv`. All tests were run under Linux (Mandrake 8.1) on bi-processors based on AMD Athlon MP 1200 MHz with 1GByte main memory. For simplicity, we fixed the initial size of the programs: While in the first series all nested programs contain 20 rules and 5 facts, the same holds in the second series regarding cardinality constraints. For simplicity, both series are further divided into subseries according to the number of variables involved; in fact, we ran subseries with 10, 15, 20,. . . , 45, 50 variables.

Since the results of all these subseries are reported at the aforecited URL, we concentrate here on one representative, given by the 25 variables subseries. This subseries comprises 260 normal form programs and 426 cardinality constraints programs. The corresponding plots are given at the above URL at `node20.html` and `node66.html`. In

Table 1. Quantitative comparison.

	Normal form		Cardinality constraints	
Feature	dlp<htl	htl<dlp	dlp<htl	htl<dlp
Size in Bytes	15	245	146	279
Size in Rules	22	238	208	217
Compilation time	10	250	182	243
Runtime	63	190	282	115

all cases, we observe, as expected, a significant increase in the size of the image under the language-preserving translation. This is confirmed in the experiments through the ratios between the size of the output and input file. Also, looking at Table 1 (where $x < y$ indicates how many times x is "larger" than y), we see that the majority of test cases has a smaller image under the structural transformations than under the language-preserving one. However, this is less significant for the cardinality constraints series than for the normal form series. Clearly, the size of the image affects also the compilation time. We thus get a similar behavior as regards compilation time, as we obtained for the size of the image (cf. Table 1). Things become more interesting when comparing the respective running times (for computing *all* answer sets of a given program; cf. the aforecited URLs). For the first time, we obtain a kind of threshold beyond which the structural transformations always outperform the language-preserving one. While this is the case for an estimated blow-up factor of 500 for the normal-form test cases, the one for the cardinality-constraint test cases is at 200. The fact that the results of all translations appear to be scattered below this threshold is confirmed by the detailed results under the threshold. Interestingly, the language-preserving translation may even be outperformed on rather small blow-up factors, as witnessed by the range of 0–50 (cf. again the above given URLs). Although these thresholds are sometimes less clear cut in the other test series, their existence seems to be a recurring feature.

References

1. P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 2003. To appear.
2. T. Janhunen, I. Niemelä, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. In *Proc. KR-00*, pages 411–419. Morgan Kaufmann, 2000.
3. N. Leone, W. Faber, G. Pfeifer, T. Eiter, G. Gottlob, C. Koch, C. Mateis, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 2003. To appear.
4. V. Lifschitz, L. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):369–389, 1999.
5. D. Pearce, V. Sarsakov, T. Schaub, H. Tompits, and S. Woltran. A polynomial translation of logic programs with nested expressions into disjunctive logic programs. In *Proc. ICLP-02*, pages 405–420. Springer, 2002.
6. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

Table of Contents

Invited Papers

Regular Papers

System Descriptions

Author Index