LNAI 6645

James P. Delgrande
Wolfgang Faber (Eds.)

Logic Programming and Nonmonotonic Reasoning

11th International Conference, LPNMR 2011
Vancouver, Canada, May 2011
Proceedings

Springer

James P. Delgrande   Wolfgang Faber (Eds.)

Logic Programming and Nonmonotonic Reasoning

11th International Conference, LPNMR 2011
Vancouver, Canada, May 16-19, 2011
Proceedings

Springer

Volume Editors

James P. Delgrande
Simon Fraser University, School of Computing Science
Burnaby, B.C., V5A 1S6, Canada
E-mail: jim@cs.sfu.ca

Wolfgang Faber
University of Calabria, Department of Mathematics
87036 Rende (CS), Italy
E-mail: wf@wfaber.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-20894-2 e-ISBN 978-3-642-20895-9
DOI 10.1007/978-3-642-20895-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011926310

CR Subject Classification (1998): I.2.3, I.2.4, F.1.1, F.4.1, D.1.6, G.2

LNCS Sublibrary: SL 7 – Artificial Intelligence

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-11) held during May 16–19, 2011 in Vancouver, British Columbia, Canada.

LPNMR is a forum for exchanging ideas on declarative logic programming, nonmonotonic reasoning and knowledge representation. The aim of the conference is to facilitate interaction between researchers interested in the design and implementation of logic-based programming languages and database systems, and researchers who work in the areas of knowledge representation and nonmonotonic reasoning. LPNMR strives to encompass theoretical and experimental studies that have led or will lead to the construction of practical systems for declarative programming and knowledge representation.

The conference program included invited talks by Chitta Baral, David Pearce, and David Poole, as well as 16 long papers (13 technical papers, 1 application description, and 2 system descriptions) and 26 short papers (16 technical papers, 3 application descriptions, and 7 system descriptions), which were selected by the Program Committee after a thorough reviewing process. The conference also hosted three workshops and the award ceremony of the Third ASP Competition, held and organized prior to the conference by Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca at the University of Calabria, Italy.

We would like to thank the members of the Program Committee and the additional reviewers for their efforts to produce fair and thorough evaluations of the submitted papers, the local Organizing Committee, especially Aaron Hunter and Jiahui Xu, and of course the authors of the scientific papers. Furthermore we are grateful to the sponsors for their generous support: *Artificial Intelligence Journal*, Pacific Institute of the Mathematical Sciences (PIMS), Assocation of Logic Programming (ALP), Simon Fraser University, and the University of Calabria. Last, but not least, we thank the people of EasyChair for providing resources and a marvelous conference management system.

March 2011

James Delgrande
Wolfgang Faber

Organization

Program Chairs

James Delgrande	Simon Fraser University, Canada
Wolfgang Faber	University of Calabria, Italy

Local Organization Chair

Aaron Hunter	Simon Fraser University, Canada

Program Committee

Jose Julio Alferes	Universidade Nova de Lisboa, Portugal
Marcello Balduccini	Kodak Research Laboratories, USA
Chitta Baral	Arizona State University, USA
Leopoldo Bertossi	Carleton University, Canada
Richard Booth	University of Luxembourg
Gerhard Brewka	Leipzig University, Germany
Pedro Cabalar	University of A Coruña, Spain
Stefania Costantini	Università di L'Aquila, Italy
Marina De Vos	University of Bath, UK
Marc Denecker	Katholieke Universiteit Leuven, Belgium
Yannis Dimopoulos	University of Cyprus
Juergen Dix	Clausthal University of Technology, Germany
Agostino Dovier	Università di Udine, Italy
Thomas Eiter	Vienna University of Technology, Austria
Esra Erdem	Sabanci University, Turkey
Michael Fink	Vienna University of Technology, Austria
Andrea Formisano	Università di Perugia, Italy
Martin Gebser	University of Potsdam, Germany
Michael Gelfond	Texas Tech University, USA
Giovambattista Ianni	University of Calabria, Italy
Tomi Janhunen	Aalto University, Finland
Antonis Kakas	University of Cyprus
Joohyung Lee	Arizona State University, USA
Nicola Leone	University of Calabria, Italy
Vladimir Lifschitz	University of Texas, USA
Fangzhen Lin	Hong Kong University of Science and Technology, P.R. China

Jorge Lobo	IBM T. J. Watson Research Center, USA
Robert Mercer	The University of Western Ontario, Canada
Alessandra Mileo	University of Milano-Bicocca, Italy
Ilkka Niemelä	Aalto University, Finland
Mauricio Osorio	Fundación Universidad de las Américas, Mexico
Ramon Otero	University of A Coruña
David Pearce	Universidad Rey Juan Carlos, Spain
Axel Polleres	National University of Ireland
Enrico Pontelli	New Mexico State University, USA
Chiaki Sakama	Wakayama University, Japan
John Schlipf	University of Cincinnati, USA
Tran Cao Son	New Mexico State University, USA
Terrance Swift	Universidade Nova de Lisboa, Portugal
Evgenia Ternovska	Simon Fraser University, Canada
Hans Tompits	Vienna University of Technology, Austria
Francesca Toni	Imperial College London, UK
Mirek Truszczynski	University of Kentucky, USA
Agustin Valverde Ramos	Universidad de Malaga, Spain
Kewen Wang	Griffith University, Australia
Stefan Woltran	Vienna University of Technology, Austria
Jia-Huai You	University of Alberta, Canada
Yan Zhang	University of Western Sydney, Australia

Additional Reviewers

Michael Bartholomew
Sandeep Chintabathina
Alessandro Dal Palù
Minh Dao Tran
Broes De Cat
Stef De Pooter
Wolfgang Dvorak
Jorge Fandino
Luis Fraga
Sarah Gaggl
Gregory Gelfond
Ricardo Gonçalves
Katsumi Inoue
Jianmin Ji
Matthias Knorr
Guohua Liu
Marco Manna
Marco Maratea
Yunsong Meng
Juan Antonio Navarro Perez
Johannes Oetsch
Emilia Oikarinen
Simona Perri
Jörg Pührer
Francesco Ricca
Torsten Schaub
Peter Schüller
Marco Sirianni
Shahab Tasharrofi
Hanne Vlaeminck
Antonius Weinzierl
Siert Wieringa
Maonian Wu
Claudia Zepeda-Cortés
Yi Zhou

Table of Contents

Logic, Probability and Computation: Foundations and Issues of Statistical Relational AI

David Poole

Department of Computer Science,
University of British Columbia,
Vancouver, BC, V6T 1Z4, Canada
poole@cs.ubc.ca
http://cs.ubc.ca/~poole/

Abstract. Over the last 25 years there has been considerable body of research into combinations of predicate logic and probability forming what has become known as (perhaps misleadingly) statistical relational artificial intelligence (StaR-AI). I overview the foundations of the area, give some research problems, proposed solutions, outstanding issues, and clear up some misconceptions that have arisen. I discuss representations, semantics, inference and learning, and provide some references to the literature. This is intended to be an overview of foundations, not a survey of research results.

Keywords: statistical relational learning, relational probabilistic models, inductive logic programming, independent choice logic, parametrized random variables.

1 Introduction

Over the last 25 years there has been a considerable body of research into combining logic and probability, evolving into what has come to be called *statistical relational AI*. Rather than giving a survey, I will motivate the issues from the bottom-up, trying to justify some choices that have been made. Laying bare the foundations will hopefully inspire others to join us in exploring the frontiers and unexplored areas.

One of the barriers to understanding this area is that it builds from multiple traditions, which often use the same vocabulary to mean different things. Common terms such as "variable", "domain", "relation", and "parameter" have come to have accepted meanings in mathematics, computing, logic and probability, but their meanings in each of these areas is different enough to cause confusion.

Both predicate logic (e.g., the first-order predicate calculus) and Bayesian probability calculus can be seen as extending the propositional calculus, one by adding relations, individuals and quantified variables, the other by allowing for measures over possible worlds and conditional queries. Relational probabilistic models[1], which form the basis of statistical relational AI can be seen as combinations of probability and predicate calculus to allow for individuals and relations as well as probabilities.

To understand the needs for such a combination, consider learning from the two datasets in Figure 1 (from [25]). Dataset (a) is the sort used in traditional supervised and unsupervised learning. Standard textbook supervised learning algorithms can learn a decision tree, a neural network, or a support vector machine to predict $UserAction$. A belief network learning algorithm can be used to learn a representation of the distribution over the features. Dataset (b), from which

Example	*Author*	*Thread*	*Length*	*WhereRead*	*UserAction*
e_1	*known*	*new*	*long*	*home*	*skips*
e_2	*unknown*	*new*	*short*	*work*	*reads*
e_3	*unknown*	*follow_up*	*long*	*work*	*skips*
e_4	*known*	*follow_up*	*long*	*home*	*skips*
...	...	...	...	...	...

(a)

Individual	Property	Value
joe	*likes*	*resort_14*
joe	*dislikes*	*resort_35*
...	...	...
resort_14	*type*	*resort*
resort_14	*near*	*beach_18*
beach_18	*type*	*beach*
beach_18	*covered_in*	*ws*
ws	*type*	*sand*
ws	*color*	*white*
...	...	...

(b)

Fig. 1. Two datasets

we may want to predict what Joe likes, is different. Many of the values in the table are meaningless names that can't be used directly in supervised learning. Instead, it is the relationship among the individuals in the world that provides the generalizations from which to learn. Learning from such datasets has been studied under the umbrella of inductive logic programming (ILP) [12,10] mainly because logic programs provide a good representation for the generalizations required to make predictions. ILP is one of the foundations of StaR-AI, as it provides a toolbox of techniques for structure learning.

One confusion about the area stems from the term "relational"; after all most of the datasets are, or can be, stored in relational databases. The techniques of

[1] Here we use this term in the broad sense, meaning any models that combine relations and probabilities.

relational probabilistic models are applicable to cases where the values in the database are names of individuals and it is the properties of the individuals and the relationship between the individuals that are modelled. It is sometimes also called multi-relational learning, as it is the interrelations that are important. This is a misnomer because, as can be seen in Figure 1 (b), it not multiple relations that cause problems (and provide opportunities to exploit structure), as a single triple relation can store any relational database (in a so-called triple-store).

The term statistical relational AI, comes from not only having probabilities and relations, but that the models are derived from data and prior knowledge.

2 Motivation

Artificial intelligence (AI) is the study of computational agents that act intelligently [25]. The basic argument for probability as a foundation of AI is that agents that act under uncertainty are gambling, and probability is the calculus of gambling in that agents who don't use probability will lose to those that do use it [33]. While there are a number of interpretations of probability, the most suitable is a Bayesian or subjective view of probability: our agents do not encounter generic events, but have to make decisions in particular circumstances, and only have access to their beliefs.

In probability theory, possible worlds are described in terms of so-called *random variables* (although they are neither random or variable). A random variable has a value in every world. We can either define random variables in terms of worlds or define worlds in terms of random variables. A random variable having a particular value is a proposition. Probability is defined in terms of a non-negative measure over sets of possible worlds that follow some very intuitive axioms.

In Bayesian probability, we make explicit assumptions and the conclusions are logical consequences of the specified knowledge and assumptions. One particular explicit assumption is the assumption of conditional independence. A Bayesian network [14] is an acyclic directed graphical model of probabilistic dependence that encapsulates the independence: a variable is conditionally independent of other variables (those that are not its descendants in the graph) given its parents in the graph. This has turned out to be a very useful assumption in practice. Undirected graphical models encapsulate the assumption that a variable is independent of other variables given its neighbours.

These motivations for probability (and similar motivations for utility) do not depend on non-relational representations.

3 Representation

Statistical relational models are typically defined in terms of parametrized random variables [20] which are often drawn in terms of plates [3]. A parametrized random variable corresponds to a predicate or a function symbol in logic. It can include logical variables (which form the parameters). In the following examples, we will write logical variables (which denote individuals) in upper case, and constants, function and predicate symbols in lower case. We assume that the logical variables are typed, where the domain of the type, the set of individuals of the type, is called the population.

Parametrized random variables are best described in terms of an example. Consider the case of diagnosing students' performance in adding multi-digit numbers of the form

$$\begin{array}{rrrr} & & x_1 & x_0 \\ + & & y_1 & y_0 \\ \hline & z_2 & z_1 & z_0 \end{array}$$

A student is given the values for the x's and the y's and provides values for the z's.

Whether a student gets the correct answer for z_i depends on x_i, y_i, the value carried in and whether she knows addition. Whether a student gets the correct carry depends on the previous x, y and carry, and whether she knowns how to carry. This dependency can be seen in Figure 2. Here $x(D, P)$ is a parametrized

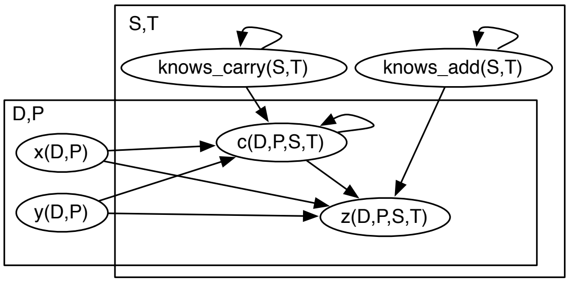


Fig. 2. Belief network with plates for multidigit addition

random variable. There is a random variable for each digit D and each problem P. A ground instance, such as $x(d_3, problem_{57})$, is a random variable that may represent the third digit of problem 57. Similarly, there is a z-variable for each digit D, problem P, student S, and time T. The plate notation can be read as duplicating the random variable for each tuple of individual the plate is parametrized by.

The basic principle used by all methods is that of *parameter sharing*: the instances of the parametrized random created by substituting constants for logical variables share the same probabilistic parameters. The various languages differ in how to specify the conditional probabilities of the variables variable given its parents, or the other parameters of the probabilistic model.

The first such languages (e.g., [8]), described the conditional probabilities directly in term of tables, and require a combination function (such as noisy-and or noisy-or) when there is a random variable parametrized by a logical variable that is a parent of a random variable that is not parametrized by the logical variable. Tables with combination functions turn out to be not a very flexible representation as they cannot represent the subtleties involved in how one random variable can depend on others.

In the above example, $c(D, P, S, T)$ depends, in part, on $c(D-1, P, S, T)$, that is, on the carry from the previous digit (and there is some other case for the first digit). A more complex example is to determine the probability that two authors are collaborators, which depends on whether they have written papers in common, or even whether they have written papers apart from each other.

To represent such examples, it is useful to be able to specify how the logical variables interact, as is done in logic programs. The independent choice logic (ICL) [18,22] (originally called probabilistic Horn abduction [15,17]) allows for arbitrary (acyclic) logic programs (including negation as failure) to be used to represent the dependency. The conditional probability tables are represented as independent probabilistic inputs to the logic program. A logic program that represents the above example is in Chapter 14 of [25]. This idea also forms the foundation for Prism [29,30], which has concentrated on learning, and for Problog [4], a project to build an efficient and flexible language.

There is also work on undirected models, exemplified by Markov logic networks [26], which have a similar notion of parametrized random variables, but the probabilities are represented as weights of first-order clauses. Such models have the advantage that they can represent cyclic dependencies, but there is no local interpretation of the parameters, as probabilistic inference relies on a global normalization.

4 Inference

Inference in these models refers to computing the posterior distribution of some variables given some evidence.

A standard way to carry out inference in such models is to try to generate and ground as few of the parametrized random variables as possible. In the ICL, the relevant ground instances can be carried out using abduction [16]. More recently, there has been work on lifted probabilistic inference [20,5,31,11], where the idea is to carry out probabilistic reasoning at the lifted level, without grounding out the parametrized random variables. Instead, we count how many of the probabilities we need, and when we need to multiply a number of identical probabilities, we can take the probability to the power of the number of individuals. Lifted inference turns out to be a very difficult problem, as the possible interactions between parametrized random variables can be very complicated.

5 Learning

The work on learning in relational probabilistic models has followed two, quite different, paths.

From a Bayesian point of view, learning is just a case of inference: we condition on all of the observations (all of the data), and determine the posterior distribution over some hypotheses or any query of interest. Starting from the work of Buntine [3], there has been considerable work in this area [9]. This work uses parametrized random variables (or the equivalent plates) and the probabilistic parameters are real-values random variables (perhaps parametrized). Dealing with real-valued variables requires sophisticated reasoning techniques often in terms of MCMC and stochastic processes. Although these methods use relational probabilistic models for learning, the representations learned are typically not relational probabilistic models.

There is a separate body of work about learning relational probabilistic models [29,7]. These typically use non-Bayesian techniques, to find the most likely models given the data (whereas the Bayesian technique is to average over all models). What is important about learning is that we want to learn general theories that can be learned before the agent know the individuals, and so before the agent knows the random variables.

It is still an open challenge to bring these two threads together, mainly because of the difficulty of inference in these complex models.

6 Actions

There is also a large body of work on representing actions. The initial work in this area was on representations, in terms of the event calculus [18] or the situation calculus [19,1][2]. This is challenging because to plan, an agent needs to be concerned about what information will be available for future decision. These models combined perception, action and utility to form first-order variants of fully-observable and partially-observable Markov decision processes.

Later work has concentrated on how to do planning with such representations either for the fully observable case [2,27] or the partially observable case [35,28]. The promise of being able to carry out lifted inference much more efficiently is slowly being realized. There is also work on relational reinforcement learning [32,34], where an agent learns what to do before knowing what individuals will be encountered, and so before it knows what random variables exist.

7 Identity and Existence Uncertainty

The previously outlined work assumes that an agent knows which individuals exist and can identify them. The problem of knowing whether two descriptions refer to the same individual is known as identity uncertainty [13]. This arises in citation matching when we need to distinguish whether two references refer to the same paper and in record linkage, where the aim is to determine if two hospital records refer to the same person (e.g., whether the current patient who is requesting drugs been at the hospital before). To solve this, we have the

[2] These two papers are interesting because they make the opposite design decisions on almost all of the design choices. For example, whether an agent knowns what situation it is in, and whether a situation implies what is true: we can't have both for a non-omniscient agent.

hypotheses of which terms refer to which individuals, which becomes combinatorially difficult.

The problem of knowing whether some individual exists is known as existence uncertainty [21]. This is challenging because when existence is false, there is no individual to refer to, and when existence is true, there may be many individuals that fit a description. We may have to know which individual a description is referring to. In general, determining the probability of an observation requires knowing the protocol for how observations were made. For example, if an agent considers a house and declares that there is a green room, the probability of this observation depends on what protocol they were using: did they go looking for a green room, did they report the colour of the first room found, did they report the type of the first green thing found, or did they report on the colour of the first thing they perceived?

8 Ontologies and Semantic Science

Data that are reliable and people care about, particularly in the sciences, are being reported using the vocabulary defined in formal ontologies [6]. The next stage in this line of research is to represent scientific hypotheses that also refer to formal ontologies and are able to make probabilistic predictions that can be judged against data [23]. This work combines all of the issues of relational probabilistic modelling as well as the problems of describing the world at multiple level of abstraction and detail, and handling multiple heterogenous data sets. It also requires new ways to think about ontologies [24], and new ways to think about the relationships beween data, hypotheses and decisions.

9 Conclusions

Real agents need to deal with their uncertainty and reason about individuals and relations. They need to learn how the world works before they have encountered all the individuals they need to reason about. If we accept these premises, then we need to get serious about relational probabilistic models. There is a growing community under the umbrella of statistical relational learning that is tackling the problems of decision making with models that refer to individuals and relations. While there have been considerable advances in the last two decades, there are more than enough problems to go around!

References

1. Bacchus, F., Halpern, J.Y., Levesque, H.J.: Reasoning about noisy sensors and effectors in the situation calculus. Artificial Intelligence 111(1-2), 171–208 (1999), `http://www.lpaig.uwaterloo.ca/~fbacchus/on-line.html`
2. Boutilier, C., Reiter, R., Price, B.: Symbolic dynamic programming for first-order MDPs. In: Proc. 17th International Joint Conf. Artificial Intelligence, IJCAI 2001 (2001)

3. Buntine, W.L.: Operations for learning with graphical models. Journal of Artificial Intelligence Research 2, 159–225 (1994)
4. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2462–2467 (2007)
5. de Salvo Braz, R., Amir, E., Roth, D.: Lifted first-order probabilistic inference. In: Getoor, L., Taskar, B. (eds.) Introduction to Statistical Relational Learning. MIT Press, Cambridge (2007), `http://www.cs.uiuc.edu/~eyal/papers/BrazRothAmir_SRL07.pdf`
6. Fox, P., McGuinness, D., Middleton, D., Cinquini, L., Darnell, J., Garcia, J., West, P., Benedict, J., Solomon, S.: Semantically-enabled large-scale science data repositories. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 792–805. Springer, Heidelberg (2006), `http://www.ksl.stanford.edu/KSL_Abstracts/KSL-06-19.html`
7. Getoor, L., Friedman, N., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: Dzeroski, S., Lavrac, N. (eds.) Relational Data Mining, pp. 307–337. Springer, Heidelberg (2001)
8. Horsch, M., Poole, D.: A dynamic approach to probabilistic inference using Bayesian networks. In: Proc. Sixth Conference on Uncertainty in AI, Boston, pp. 155–161 (July 1990)
9. Jordan, M.I.: Bayesian nonparametric learning: Expressive priors for intelligent systems. In: Dechter, R., Geffner, H., Halpern, J.Y. (eds.) Heuristics, Probability and Causality: A Tribute to Judea Pearl, pp. 167–186. College Publications (2010)
10. Lavrac, N., Dzeroski, S.: Inductive Logic Programming: Techniques and Applications. Ellis Horwood, NY (1994)
11. Milch, B., Zettlemoyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted probabilistic inference with counting formulas. In: Proceedings of the Twenty Third Conference on Artificial Intelligence, AAAI (2008), `http://people.csail.mit.edu/lpk/papers/mzkhk-aaai08.pdf`
12. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19(20), 629–679 (1994)
13. Pasula, H., Marthi, B., Milch, B., Russell, S., Shpitser, I.: Identity uncertainty and citation matching. In: NIPS, vol. 15 (2003)
14. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Mateo (1988)
15. Poole, D.: Representing diagnostic knowledge for probabilistic Horn abduction. In: Proc. 12th International Joint Conf. on Artificial Intelligence (IJCAI 1991), Sydney, pp. 1129–1135 (1991)
16. Poole, D.: Logic programming, abduction and probability: A top-down anytime algorithm for computing prior and posterior probabilities. New Generation Computing 11(3-4), 377–400 (1993)
17. Poole, D.: Probabilistic Horn abduction and Bayesian networks. Artificial Intelligence 64(1), 81–129 (1993)
18. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artificial Intelligence 94, 7–56 (1997), `http://cs.ubc.ca/~poole/abstracts/icl.html`; special issue on economic principles of multi-agent systems
19. Poole, D.: Decision theory, the situation calculus and conditional plans. Electronic Transactions on Artificial Intelligence 2(1-2) (1998), `http://www.etaij.org`

20. Poole, D.: First-order probabilistic inference. In: Proc. Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, pp. 985–991 (2003)
21. Poole, D.: Logical generative models for probabilistic reasoning about existence, roles and identity. In: 22nd AAAI Conference on AI (AAAI 2007) (July 2007), `http://cs.ubc.ca/~poole/papers/AAAI07-Poole.pdf`
22. Poole, D.: The independent choice logic and beyond. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S.H. (eds.) Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 222–243. Springer, Heidelberg (2008), `http://cs.ubc.ca/~poole/papers/ICL-Beyond.pdf`
23. Poole, D., Smyth, C., Sharma, R.: Semantic science: Ontologies, data and probabilistic theories. In: da Costa, P.C.G., d'Amato, C., Fanizzi, N., Laskey, K.B., Laskey, K.J., Lukasiewicz, T., Nickles, M., Pool, M. (eds.) URSW 2005 - 2007. LNCS (LNAI), vol. 5327, pp. 26–40. Springer, Heidelberg (2008), `http://cs.ubc.ca/~poole/papers/SemSciChapter2008.pdf`
24. Poole, D., Smyth, C., Sharma, R.: Ontology design for scientific theories that make probabilistic predictions. IEEE Intelligent Systems 24(1), 27–36 (2009), `http://www2.computer.org/portal/web/computingnow/2009/0209/x1poo`
25. Poole, D.L., Mackworth, A.K.: Artificial Intelligence: foundations of computational agents. Cambridge University Press, New York (2010), `http://artint.info`
26. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62, 107–136 (2006)
27. Sanner, S., Boutilier, C.: Approximate linear programming for first-order MDPs. In: Proceedings of the Twenty-first Conference on Uncertainty in Artificial Intelligence (UAI 2005), Edinburgh, pp. 509–517 (2005)
28. Sanner, S., Kersting, K.: Symbolic dynamic programming for first-order POMDPs. In: Proc. AAAI 2010 (2010)
29. Sato, T., Kameya, Y.: PRISM: A symbolic-statistical modeling language. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), pp. 1330–1335 (1997)
30. Sato, T., Kameya, Y.: New advances in logic-based probabilistic modeling by PRISM. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S.H. (eds.) Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 118–155. Springer, Heidelberg (2008), `http://www.springerlink.com/content/1235t75977x62038/`
31. Singla, P., Domingos, P.: Lifted first-order belief propagation. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, pp. 1094–1099 (2008)
32. Tadepalli, P., Givan, R., Driessens, K.: Relational reinforcement learning: An overview. In: Proc. ICML Workshop on Relational Reinforcement Learning (2004)
33. Talbott, W.: Bayesian epistemology. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy (Fall 2008), `http://plato.stanford.edu/archives/fall2008/entries/epistemology-bayesian/`
34. van Otterlo, M.: The Logic of Adaptive Behavior - Knowledge Representation and Algorithms for Adaptive Sequential Decision Making under Uncertainty in First-Order and Relational Domains. IOS Press, Amsterdam (2009), `http://people.cs.kuleuven.be/~martijn.vanotterlo/phdbook_vanOtterlo_v2010a.pdf`
35. Wang, C., Khardon, R.: Relational partially observable MDPs. In: Proc. AAAI 2010 (2010)

Foundations and Extensions of Answer Set Programming: The Logical Approach

David Pearce

Departamento de Inteligencia Artificial
Universidad Politécnica de Madrid

Overview

Answer Set Programming, or ASP, is now becoming well-established as a declarative approach to problem-solving in AI and in an increasing number of practical, application domains. While a significant part of ASP research is devoted to producing and applying faster and more user-friendly solvers, there is also a growing interest in studying extensions of the basic language of ASP. Additions to the language that are currently being developed include function symbols, temporal and causal operators, as well as devices to deal with aggregates, preferences, ontologies, resources, and many others.

In this enterprise of building and implementing language extensions it is hard to detect any overarching, unifying or dominant methodology. Perhaps it is good, as a famous Austrian philosopher once proposed, that "anything goes"[1]. On the other hand perhaps there are some methods, guidelines or heuristics that we can put to good use in a systematic way when we are analysing and developing ASP languages. In this talk I will describe and argue for one kind of methodology in particular. I call it the *logical approach* to the foundations and extensions of ASP. It is definitely not an approach that would have been endorsed by that famous Austrian philosopher. He took the view that logic is a constraining mechanism, a straitjacket that inhibits the free flow of new ideas, thus being detrimental to the advancement of science. However, I will argue instead that logic is a liberating device that when properly used can be a source of inspiration, information and unification.

There is a conventional or *received* view of answer set programs. According to this, very roughly, a program consists of a set of rules comprising a head and a body, each composed of (sets of) atoms or literals. To these, various syntactic devices may be added to express additional properties or narrow down permitted interpretations of the program. The semantics, or the allowed interpretations of the program, are specified operationally by means of a fixpoint construction or equation whose solutions are the intended answer sets. We can contrast this with the logical view of answer set programs. According to this an answer set program is a logical *theory* composed of sentences in some propositional, predicate or other language that may, eg, contain modal, temporal or epistemic operators. Whatever the syntactic restrictions, the language is fully recursive and the basic

[1] See eg. P. Feyerabend, *Against Method*, VLB, 1975.

semantics is given by standard, inductive truth conditions. The intended models of the program are a special set of minimal models whose definition depends on the particular language or 'logic' at hand.

Most of the time these two views of ASP lead to the same result or output in the sense that they agree in producing the same or equivalent answer sets for a given program or theory. Sometimes we can even translate one view into the other. But they are not fully equivalent and do not yield equivalent methodologies.

In the talk I will elaborate further on the logical view and examine several kinds of language extensions of ASP from this perspective. A decisive advantage of the logical approach is that it can build upon results, techniques and heuristics that are useful both for analysing and computing as well as for designing new features and languages. These techniques include proof theory and logical inference, replacement theorems and logical equivalence, the analysis of normal forms such as prenex and Skolem forms, metatheorems such as Interpolation, as well as model theory and complexity theory in general.

Another feature of the logical approach is that it may help to give us 'smooth' interfaces between different types of calculi, as when we extend a propositional language with modal operators or when we combine two different logics. I will also discuss examples where the logical approach may provide natural and convincing criteria for choosing between competing, alternative formalisations of a given concept or language extension.

Lessons from Efforts to Automatically Translate English to Knowledge Representation Languages

Chitta Baral

Faculty of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-8809
chitta@asu.edu

Abstract. Our long term goal is to develop systems that can "understand" natural language text. By "understand" we mean that the system can take natural language text as input and answer questions with respect to that text. A key component in building such systems is to be able to translate natural language text into appropriate knowledge representation (KR) languages. Our approach to achieve that is inspired by Montagues path breaking thesis (1970) of viewing English as a formal language and by the research in natural language semantics. Our approach is based on PCCG (Probabilistic Combinatorial Categorial Grammars), λ-calculus and statistical learning of parameters. In an initial work, we start with an initial vocabulary consisting of λ-calculus representations of a small set of words and a training corpus of sentences and their representation in a KR language. We develop a learning based system that learns the λ-calculus representation of words from this corpus and generalizes it to words of the same category. The key and novel aspect in this learning is the development of Inverse Lambda algorithms which when given λ-expressions β and γ can come up with an α such that application of α to β (or β to α) will give us γ. We augment this with learning of weights associated with multiple meanings of words. Our current system produces improved results on standard corpora on natural language interfaces for robot command and control and database queries. In a follow-up work we are able to use patterns to make guesses regarding the initial vocabulary. This together with learning of parameters allow us to develop a fully automated (without any initial vocabulary) way to translate English to designated KR languages. In an on-going work we use Answer Set Programming as the target KR language and focus on (a) solving combinatorial puzzles that are described in English and (b) answering questions with respect to a chapter in a ninth grade biology book. The systems that we are building are good examples of integration of results from multiple sub-fields of AI and computer science, viz.: machine learning, knowledge representation, natural language processing, λ-calculus (functional programming) and ontologies. In this presentation we will describe our approach and our system and elaborate on some of the lessons that we have learned from this effort.

Modularity of P-Log Programs

Carlos Viegas Damásio and João Moura

CENTRIA, Departamento de Informática
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

Abstract. We propose an approach for modularizing P-log programs and corresponding compositional semantics based on conditional probability measures. We do so by resorting to Oikarinen and Janhunen's definition of a logic program module and extending it to P-log by introducing the notions of input random attributes and output literals. For answering to P-log queries our method does not imply calculating all the stable models (possible worlds) of a given program, and previous calculations can be reused. Our proposal also handles probabilistic evidence by conditioning (observations).

Keywords: P-log, Answer Set Programming, Modularization, Probabilistic Reasoning.

1 Introduction and Motivation

The P-log language [3] has emerged as one of the most flexible frameworks for combining probabilistic reasoning with logical reasoning, in particular, by distinguishing acting (doing) from observations and allowing non-trivial conditioning forms [3,4]. The P-log languages is a non-monotonic probabilistic logic language supported by two major formalisms, namely Answer Set Programming [7,11,12] for declarative knowledge representation and Causal Bayesian Networks [15] as its probabilistic foundation. In particular, ordinary Bayesian Networks can be encoded in P-log. The relationships of P-log to other alternative uncertainty knowledge representation languages like [10,16,17] have been studied in [3]. Unfortunately, the existing current implementations of P-log [1,8] have exponential best case complexity, since they enumerate all possible models, even though it is known that for singly connected Bayesian Networks (polytrees) reasoning can be performed in polynomial time [14].

The contribution of this paper is the definition of modules for the P-log language, and corresponding compositional semantics as well as its probabilistic interpretation. The semantics relies on a translation to logic program modules of Oikarinen and Janhunen [13]. With this appropriate notion of P-log modules is possible to obtain possible worlds incrementally, and this can be optimized for answering to probabilistic queries in polynomial time for specific cases, using techniques inspired in the variable elimination algorithm [20].

The rest of the paper is organized as follows. Section 2 briefly summarizes P-log syntax and semantics, as well as the essential modularity results for answer set programming. Next, Section 3 is the core of the paper defining modules for P-log language and its translation into ASP modules. The subsequent section presents the module theorem and a discussion of the application of the result to Bayesian Networks. We conclude with final remarks and foreseen work.

2 Preliminaries

In this section, we review the syntax and semantics of P-log language [3], and illustrate it with an example encoding a Bayesian Network. Subsequently, the major results regarding composition of answer set semantics are presented [13]. The reader is assumed to have familiarity with (Causal) Bayesian Networks [15] and good knowledge of answer set programming. A good introduction to Bayesian Networks can be found in [19], and to answer set programming in [2,12].

2.1 P-Log Programs

P-log is a declarative language [3], based on a logic formalism for probabilistic reasoning and action, that uses answer set programming (ASP) as its logical foundation and Causal Bayesian Networks (CBNs) as its probabilistic foundation. P-log is a complex language to present in a short amount of space, and the reader is referred to [3] for full details. We will try to make the presentation self-contained for this paper, abbreviating or even neglecting the irrelevant parts, and follows closely [3].

P-log syntax. A probabilistic logic program (P-log program) Π consists of (i) a sorted signature, (ii) a declaration part, (iii) a regular part, (iv) a set of random selection rules, (v) a probabilistic information part, and (vi) a set of observations and actions. Notice that the first four parts correspond to the actual stable models' generation, and the last two define the probabilistic information.

The declaration part defines a sort c by explicitly listing its members with a statement $c = \{x_1, \ldots, x_n\}$, or by defining a unary predicate c in a program with a single answer set. An attribute a with n parameters is declared by a statement $a : c_1 \times \ldots \times c_n \rightarrow c_0$ where each c_i is a sort ($0 \leq i \leq m$); in the case of an attributes with no parameter the syntax $a : c_0$ may be used. By $range(a)$ we denote the set of elements of sort c_0. The sorts can be understood as domain declarations for predicates and attributes used in the program, for appropriate typing of argument variables.

The regular part of a P-log program is just a set of Answer Set Programming rules (without disjunction) constructed from the usual literals in answer set programming plus attribute literals of the form $a(\bar{t}) = t_0$ (including strongly negated literals), where $\bar{t}$ is a vector of n terms and t_0 is a term, respecting the corresponding sorts in the attribute declaration. Given a sorted signature Σ we denote by $Lit(\Sigma)$ the set of literals in Σ (i.e. Σ-literals) excluding all unary atoms $c_i/1$ used for specifying sorts.

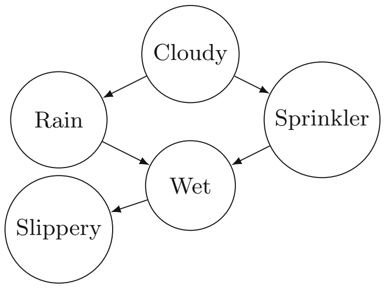
Random selection rules define random attributes and possible values for them through statements of the form $[r]\ random\ \big(a(\bar{t}) : \{X : p(X)\}\big) \leftarrow B$, expressing that if B holds then the value of $a(\overline{t})$ is selected at random from the set $\{X : p(X)\} \cap range(a)$ by experiment r, unless this value is fixed by a deliberate action, with r being a term uniquely identifying the rule. The concrete probability distribution for random attributes is conveyed by the probabilistic information part containing *pr-atoms* (probability atoms), of the form $pr_r(a(\overline{t}) = y|_c B) = v$ stating that if the value of $a(\overline{t})$ is fixed by experiment r and B holds, then the probability that r causes $a(\overline{t}) = y$ is v, with $v \in [0, 1]$. The condition B is a conjunction of literals or the default negation (not) of literals.

Finally, observations and actions are statements of the form $obs(l)$ and $do(a(\overline{t}) = y)$, respectively, where l is an arbitrary literal of the signature.

Example 1 (Wet Grass). Suppose that there are two events which could cause grass to be wet: either the sprinkler is on, or it is raining. Furthermore, suppose that the rain has a direct effect on the use of the sprinkler (namely that when it rains, the sprinkler is usually not turned on). Furthermore, cloudy sky affects whether the sprinklers are on and obviously if it is raining or not. Finally, notice that, the grass being wet affects it being slippery. Then the situation can be modeled with a Bayesian network (shown in the diagram). All random variables are boolean and have no parameters; also notice how the conditional probability tables (CPTs) are encoded with pr-atoms as well as causal dependencies among random attributes. The P-log semantics will take care of completing the CPTs assuming a uniform distribution for the remaining attribute values

$boolean = \{t, f\}.$
$cloudy : boolean.$
$rain : boolean.$
$sprinkler : boolean.$
$wet : boolean.$
$slippery : boolean.$

$dangerous \leftarrow slippery = t.$



$[rc]\, random(cloudy, \{X : boolean(X)\}).$ $[rr]\, random(rain, \{X : boolean(X)\}).$
$[rsk]\, random(sprinkler, \{X : boolean(X)\}).$ $[rw]\, random(wet, \{X : boolean(X)\}).$
$[rsl]\, random(slippery, \{X : boolean(X)\}).$

$pr_{rr}(rain = t \mid_c cloudy = f) = 0.2.$ $pr_{rr}(rain = t \mid_c cloudy = t) = 0.8.$
$pr_{rsk}(sprinkler = t \mid_c cloudy = f) = 0.5.$ $pr_{rsk}(sprinkler = t \mid_c cloudy = t) = 0.1.$
$pr_{rsl}(slippery = t \mid_c wet = f) = 0.1.$ $pr_{rsl}(slippery = t \mid_c wet = t) = 0.9.$
$pr_{rw}(wet = t \mid_c sprinkler = f, rain = f) = 0.0.$
$pr_{rw}(wet = t \mid_c sprinkler = f, rain = t) = 0.9.$
$pr_{rw}(wet = t \mid_c sprinkler = t, rain = f) = 0.9.$
$pr_{rw}(wet = t \mid_c sprinkler = t, rain = t) = 0.99.$ $obs(sprinkler = t).$

Fig. 1. Bayesian Network encoded in P-log

(e.g. $cloudy = false$ will have probability 0.5). Rules can be used to extract additional knowledge from the random variables (e.g. the *dangerous* rule). In particular we will be able to query the program to determine the probability $\mathbf{P}(dangerous|sprinkler = t)$.

P-log semantics. The semantics of a P-log program Π is given by a collection of the possible sets of beliefs of a rational agent associated with Π, together with their probabilities. We refer to these sets of beliefs as possible worlds of Π. Note that due to the restriction on the signature of P-log programs the authors enforce (all sorts are finite), possible worlds of Π are always finite. The semantics is defined in two stages. First we will define a mapping of the logical part of Π into its Answer Set Programming counterpart, $\tau(\Pi)$. The answer sets of $\tau(\Pi)$ will play the role of possible worlds of Π. The probabilistic part of Π is used to define a measure over the possible worlds, and from these the probabilities of formulas can be determined. The set of all possible worlds of Π will be denoted by $\Omega(\Pi)$.

The Answer Set Program $\tau(\Pi)$ is defined in the following way, where capital letters are variables being grounded with values from the appropriate sort; to reduce overhead we omit the sort predicates for variables in the program rules. It is also assumed that any attribute literal $a(\bar{t}) = y$ is replaced consistently by the predicate $a(\bar{t}, y)$ in the translated program $\tau(\Pi)$, constructed as follows:

τ_1: For every sort $c = \{x_1, \dots, x_n\}$ of Π, $\tau(\Pi)$ contains $c(x_1), \dots, c(x_n)$. For any remaining sorts defined by an ASP program T in Π, then $T \subseteq \tau(\Pi)$.

τ_2: Regular part:

- **(a)** For each rule r in the regular part of Π, $\tau(\Pi)$ contains the rule obtained by replacing each occurrence of an atom $a(\bar{t}) = y$ in r by $a(\bar{t}, y)$.
- **(b)** For each attribute term $a(\bar{t})$, $\tau(\Pi)$ contains $\neg a(\bar{t}, Y_1)$:$-a(\bar{t}, Y_2)$, $Y_1 \neq Y_2$ guaranteeing that in each answer set $a(\bar{t})$ has at most one value.

τ_3: Random selections:

- **(a)** For an attribute $a(\bar{t})$, we have the rule: $intervene(a(\bar{t}))$:$-do(a(\bar{t}, Y))$. Intuitively, the value of $a(\bar{t})$ is fixed by a deliberate action, i.e. $a(\bar{t})$ will not be considered random in possible worlds satisfying $intervene(a(\bar{t}))$.
- **(b)** Random selection $[r]$ $random\left(a(\bar{t}) : \{X : p(X)\}\right) \leftarrow B$ is translated into rule $1\{$ $a(\bar{t}, Z)$: $poss(r, a(\bar{t}), Z)$ $\}1$:$-B$, not $intervene(a(\bar{t}))$ and $poss(r, a(\bar{t}), Z)$:$-c_0(Z), p(Z), B$,not $intervene(a(\bar{t}))$ with $range(a) = c_0$.

τ_4: Each pr-atom $pr_r(a(\bar{t}) = y|_c B) = v$ is translated into the following rule $pa(r, a(\bar{t}, y), v)$:$-poss(r, a(\bar{t}), y), B$ of $\tau(\Pi)$ with $pa/3$ a reserved predicate.

τ_5: $\tau(\Pi)$ contains actions and observations of Π.

τ_6: For each Σ-literal l , $\tau(\Pi)$ contains the constraint :$-obs(l), not\ \ l$.

τ_7: For each atom $a(t) = y$, $\tau(\Pi)$ contains the rule $a(\bar{t}, y)$:$-do(a(\bar{t}, y))$.

In the previous construction, the two last rules guarantee respectively that no possible world of the program fails to satisfy observation l, and that the atoms made true by the action are indeed true. The introduction of the reserved predicates $poss/3$ and $pa/3$ is a novel contribution to the transformation, and simplifies the presentation of the remaining details of the semantics.

P-log semantics assigns a probability measure for each world W, i.e. answer set, of $\tau(\Pi)$ from the causal probability computed deterministically from instances of predicates $poss/3$ and $pa/3$ true in the world. Briefly, if an atom $pa(r, a(\bar{t}, y), v)$ belongs to W then the causal probability $\mathbf{P}(W, a(\bar{t}) = y)$ is v, i.e. the assigned probability in the model. The possible values for $a(\bar{t})$ are collected by $poss(r, a(\bar{t}, y_k))$ instances true in W, and P-log semantics assigns a (default) causal probability for non-assigned values, by distributing uniformly the non-assigned probability among these non-assigned values. The details to make this formally precise are rather long [3] but for our purpose it is enough to understand that for each world W the causal probability $\sum_{y \in range(a)} \mathbf{P}(W, a(\bar{t}) = y) = 1.0$, for each attribute term with at least a possible value. These probability calculations can be encoded in ASP rules with aggregates ($\#sum$ and $\#count$), making use of only $pa/3$ and $poss/3$ predicates.

Example 2. Consider the P-log program of Example 1. This program has 16 possible worlds (notice that $sprinkler = t$ is observed). One possible world is W_1 containing:

$$\begin{array}{cccccc} cloudy(f) & rain(f) & wet(t) & sprinkler(t) & slippery(t) & obs(sprinkler(t)) \\ \neg cloudy(t) & \neg rain(t) & \neg wet(f) & \neg sprinkler(f) & \neg slippery(f) & dangerous \end{array}$$

Furthermore the following probability assignment atoms are true in that model:

$$\begin{array}{lll} poss(rc, cloudy, t) & poss(rc, cloudy, f) & \\ poss(rr, rain, t) & poss(rr, rain, f) & pa(rr, rain(t), 0.2) \\ poss(rsk, sprinkler, t) & poss(rsk, sprinkler, f) & pa(rsk, sprinkler(t), 0.5) \\ poss(rsl, slippery, t) & poss(rsl, slippery, f) & pa(rsl, slippery(t), 0.9) \\ poss(rw, wet, t) & poss(rw, wet, f) & pa(rw, wet(t), 0.9) \end{array}$$

which determines the following causal probabilities in the model

$$\begin{array}{ll} \mathbf{P}(W_1, cloudy = t) = \frac{1.0-0.0}{2} = 0.5 & \mathbf{P}(W_1, cloudy = f) = \frac{1.0-0.0}{2} = 0.5 \\ \mathbf{P}(W_1, rain = t) = 0.2 & \mathbf{P}(W_1, rain = f) = \frac{1.0-0.2}{1} = 0.8 \\ \mathbf{P}(W_1, sprinkler = t) = 0.5 & \mathbf{P}(W_1, sprinkler = f) = \frac{1.0-0.5}{1} = 0.5 \\ \mathbf{P}(W_1, wet = t) = 0.9 & \mathbf{P}(W_1, wet = f) = \frac{1.0-0.9}{1} = 0.1 \\ \mathbf{P}(W_1, slippery = t) = 0.9 & \mathbf{P}(W_1, slippery = f) = \frac{1.0-0.9}{1} = 0.1 \end{array}$$

The authors define next the measure μ_Π induced by a P-log program Π:

Definition 1 (Measure). *Let W be a possible world of a P-log program Π. The unnormalized probability of W induced by Π is $\hat{\mu}_\Pi(W) = \prod_{a(\bar{t},y) \in W} \boldsymbol{P}(W, a(\bar{t}) = y)$ where the product is taken over atoms for which $\boldsymbol{P}(W, a(\bar{t}) = y)$ is defined.*

If Π is a P-log program having at least one possible world with nonzero unnormalized probability, then the measure, $\mu_\Pi(W)$, of a possible world W induced by Π is the normalized probability of W divided by the sum of the unnormalized *probabilities of all possible worlds of Π, i.e., $\mu_\Pi(W) = \frac{\hat{\mu}_\Pi(W)}{\sum_{W_i \in \Omega} \hat{\mu}_\Pi(W_i)}$. When the program Π is clear from the context we may simply write $\hat{\mu}$ and μ instead of $\hat{\mu}_\Pi$ and μ_Π respectively.*

Example 3. For world W_1 of Example 2 we obtain that:

$$\begin{aligned}\hat{\mu}(W_1) = &\mathbf{P}(W_1, cloudy = f) \times \mathbf{P}(W_1, rain = f) \times \mathbf{P}(W_1, wet = t) \times \\ &\mathbf{P}(W_1, sprinkler = t) \times \mathbf{P}(W_1, slippery = t) = \\ = &0.5 \times 0.8 \times 0.9 \times 0.5 \times 0.1 = 0.018\end{aligned}$$

Since the sum of the unconditional probability measure of all the sixteen worlds of the P-log program is 0.3 then we obtain that $\mu(W_1) = 0.06$.

The truth and falsity of propositional formulas with respect to possible worlds are defined in the standard way. A formula A, true in W, is denoted by $W \vdash A$.

Definition 2 (Probability). *Suppose Π is a P-log program having at least one possible world with nonzero unnormalized probability. The probability, $P_\Pi(A)$, of a formula A is the sum of the measures of the possible worlds of Π on which A is true, i.e. $P_\Pi(A) = \sum_{W \vdash A} \mu_\Pi(W)$.*

Conditional probability in P-log is defined in the usual way by $P_\Pi(A|B) = P_\Pi(A \wedge B)/P_\Pi(B)$ whenever $P_\Pi(B) \neq 0$, where the set B stands for the conjunction of its elements. Moreover, under certain consistency conditions on P-log programs Π, formulas A, and a set of literals B such that $P_\Pi(B) \neq 0$, it is the case that $P_\Pi(A|B) = P_{\Pi \cup obs(B)}(A)$. See the original work [3] where the exact consistency conditions are stated, which are assumed to hold subsequently.

2.2 Modularity in Answer Set Programming

The modular aspects of Answer Set Programming have been clarified in recent years [13,5] describing how and when can two program parts (modules) be composed together. In this paper, we will make use of Oikarinen and Janhunen's logic program modules defined in analogy to [6]:

Definition 3 (Module [13]). *A logic program module $\mathbb{P}$ is $\langle R, I, O, H \rangle$:*

1. *R is a finite set of rules;*
2. *I, O, and H are pairwise disjoint sets of input, output, and hidden atoms;*
3. *$At(R) \subseteq At(\mathbb{P})$ defined by $At(\mathbb{P}) = I \cup O \cup H$; and*
4. *$head(R) \cap I = \emptyset$.*

The atoms in $At_v(\mathbb{P}) = I \cup O$ are considered to be visible and hence accessible to other modules composed with $\mathbb{P}$ either to produce input for $\mathbb{P}$ or to make use of the output of $\mathbb{P}$. The hidden atoms in $At_h(\mathbb{P}) = H = At(\mathbb{P}) \backslash At_v(\mathbb{P})$ are used to formalize some auxiliary concepts of $\mathbb{P}$ which may not be sensible for other modules but may save space substantially. The condition $head(R) \cap I = \emptyset$ ensures that a module may not interfere with its own input by defining input atoms of I in terms of its rules. Thus, input atoms are only allowed to appear as conditions in rule bodies. The answer set semantics is generalized to cover modules by introducing a generalization of the Gelfond-Lifschitz's fixpoint definition. In addition to negative default literals (i.e. not l), also literals involving input atoms are used in the stability condition.

Definition 4. *An interpretation $M \subseteq At(\mathbb{P})$ is an answer set of an ASP program module $\mathbb{P} = \langle R, I, O, H \rangle$, if and only if $M = LM\left(R_I^M \cup \{a. | a \in M \cap I\}\right)$*[1] *The set of answer sets of module $\mathbb{P}$ is denoted by $AS(\mathbb{P})$.*

Given two modules $\mathbb{P}_1 = \langle R_1, I_1, O_1, H_1 \rangle$ and $\mathbb{P}_2 = \langle R_2, I_2, O_2, H_2 \rangle$, their composition $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined when their output signatures are disjoint, that is, $O1 \cap O2 = \emptyset$, and they respect each others hidden atoms, i.e. $H_1 \cap At(\mathbb{P}_2) = \emptyset$ and $H_2 \cap At(\mathbb{P}_1) = \emptyset$. Then their composition is $\mathbb{P}_1 \oplus \mathbb{P}_2 = \langle R_1 \cup R_2, (I_1 \cup I_2)\backslash(O_1 \cup O_2), O_1 \cup O_2, H_1 \cup H_2 \rangle$. However, the conditions given for $\oplus$ are not enough to guarantee compositionality in the case of stable models:

Definition 5. *Given modules $\mathbb{P}_1, \mathbb{P}_2 \in M$, their join is $\mathbb{P}_1 \sqcup \mathbb{P}_2 = \mathbb{P}_1 \oplus \mathbb{P}_2$ provided that (i) $\mathbb{P}_1 \oplus \mathbb{P}_2$ is defined and (ii) $\mathbb{P}_1$ and $\mathbb{P}_2$ are mutually independent*[2].

Theorem 1 (The module theorem). *If $\mathbb{P}_1, \mathbb{P}_2$ are modules such that $\mathbb{P}_1 \sqcup \mathbb{P}_2$ is defined, then $AS(\mathbb{P}_1 \sqcup \mathbb{P}_2) = AS(\mathbb{P}_1) \bowtie AS(\mathbb{P}_2)$, where $AS(\mathbb{P}_1) \bowtie AS(\mathbb{P}_2) = \{M_1 \cup M_2 \mid M_1 \in AS(\mathbb{P}_1), M_2 \in AS(\mathbb{P}_2),\ \text{and}\ M_1 \cap At_v(\mathbb{P}_2) = M_2 \cap At_v(\mathbb{P}_1)\}$.*

The module theorem also straightforwardly generalizes for a collection of modules because the module union operator $\sqcup$ is commutative, associative, and idempotent [13].

3 P-Log Modules

In this section, we define the notion of P-log modules and its semantics via a translation into logic program modules. Its probabilistic interpretation is provided by a conditional probability measure. In what follows, we assume that different modules may have different sorted signatures Σ.

Definition 6. *A P-log module $\mathfrak{P}$ over Σ is a structure $\langle \Pi, Rin, Rout \rangle$ such that:*

1. *Π is a P-log program (possibly with observations and actions);*
2. *Rin is a set of ground attribute literals $a(\bar{t}) = y$, of random attributes declared in Π such that $y \in range(a)$;*
3. *$Rout$ is a set of ground Σ-literals, excluding attribute literals $a(\bar{t}) = y \in Rin$;*
4. *Π does not contain rules for any attribute $a(\bar{t})$ occurring in attribute literals of Rin, i.e. no random selection rule for $a(\bar{t})$, no regular rule with head $a(\bar{t}) = y$ nor a pr-atom for $a(\bar{t}) = y$.*

[1] Note that R_I^M is a reduct allowing weighted and choice rules, and LM is an operator returning the least model of the positive program argument.

[2] There are no positive cyclic dependencies among rules in different modules.

The notion of P-log module is quite intuitive. First, the P-log program specifies the possible models and corresponding probabilistic information as before, and may include regular rules. However, the P-log module is parametric on a set of attribute terms *Rin*, which can be understood as the module's parent random variables. *Rout* specifies the random attributes which are visible as well as other derived logical conclusions. The last condition ensures that there is no interference between input and output random attributes.

The semantics of a P-log module is defined again in two stages. The possible worlds of a P-log module are obtained from the Answer Sets of a corresponding logic programming module. For simplifying definitions we assume that the isomorphism of attribute literals $a(\overline{t}) = y$ with $a(\overline{t}, y)$ instances is implicitly applied when moving from the P-log side to ASP side, and vice-versa.

Definition 7. *Consider a P-log module* $\mathfrak{P} = \langle \Pi, Rin, Rout \rangle$ *over signature* Σ, *and let* $\mathbb{P}(\mathfrak{P}) = \langle R_\mathfrak{P}, I_\mathfrak{P}, O_\mathfrak{P}, H_\mathfrak{P} \rangle$ *be the corresponding ASP module such that:*

1. $R_\mathfrak{P}$ *is* $\tau(\Pi) \cup \{{:-}a(\overline{t}, y_1), a(\overline{t}, y_2) \mid a(\overline{t}) = y_1$ *and* $a(\overline{t}) = y_2$ *in Rin s.t.* $y_1 \neq y_2\} \cup \{{:-}$ *not* $hasval_\mathfrak{P}(a(\overline{t}))\} \cup \{hasval_\mathfrak{P}(a(\overline{t})){:-}a(\overline{t}, y) \mid a(\overline{t}) = y \in Rin\}$, *where predicates defining sorts have been renamed apart;*
2. *The set of input atoms* $I_\mathfrak{P}$ *of* $\mathbb{P}(\mathfrak{P})$ *is Rin.*
3. *The set of output atoms* $O_\mathfrak{P}$ *of* $\mathbb{P}(\mathfrak{P})$ *is Rout union all instances of* $pa/3$ *and* $poss/3$ *predicates of random attributes in Rout;*
4. *The set of hidden atoms* $H_\mathfrak{P}$ *of* $\mathbb{P}(\mathfrak{P})$ *is formed by* $hasval_\mathfrak{P}/1$ *instances for attribute literals in Rin, the* Σ*-literals not included in the output or input atoms of* $\mathbb{P}(\mathfrak{P})$, *with all sort predicates renamed apart.*

The possible models of $\mathfrak{P}$ *are* $\Omega(\mathfrak{P}) = \{M \cap (Rin \cup Rout) \mid M \in AS(\mathbb{P}(\mathfrak{P}))\}$. *The name* $hasval_\mathfrak{P}$ *is local to* $\mathfrak{P}$ *and not occurring elsewhere.*

The necessity of having sort predicates renamed apart is essential to avoid name clashes between different modules using the same sort attributes. Equivalently, the program can be instantiated, and all sort predicates removed. The extra integrity constraints in $R_\mathfrak{P}$ discard models where a random attribute has not exactly one assigned value. The set of input atoms in $\mathfrak{P}$ is formed by the random attribute literals in *Rin*. The set of output atoms includes all the instances of $pa/3$ and $poss/3$ in order to be possible to determine the causal probabilities in each model. By convention, all the remaining literals are hidden. A significant difference to the ordinary ASP modules is that the set of possible models are projected with respect to the visible literals, discarding hidden information in the models. The semantics of a P-log module is defined by probabilistic conditional measures:

Definition 8. *Consider a P-log module* $\mathfrak{P} = \langle \Pi, Rin, Rout \rangle$ *over signature* Σ. *Let* E *be any subset of* $Rin \cup Rout$, *and* W *be a possible world of P-log module* $\mathfrak{P}$. *If* $E \subseteq W$ *then the conditional unnormalized probability of* W *given* E *induced by* $\mathfrak{P}$ *is*

$$\hat{\mu}_\mathfrak{P}(W|E) = \sum_{M_i \in AS(\mathbb{P}(\mathfrak{P}))\ s.t.\ M_i \cap (Rin \cup Rout) = W} \quad \prod_{a(\overline{t}, y) \in M_i} \boldsymbol{P}(M_i, a(\overline{t}) = y)$$

where the product is taken over atoms for which $\boldsymbol{P}(M_i, a(\bar{t}) = y)$ *is defined in* M_i*. Otherwise,* $E \nsubseteq W$ *and we set* $\hat{\mu}_{\mathfrak{P}}(W|E) = 0.0$*.*

If there is at least one possible world with nonzero unnormalized conditional probability, for a particular E*, then the conditional probability measure* $\mu_{\mathfrak{P}}(.|E)$ *is determined, and* $\mu_{\mathfrak{P}}(W|E)$ *for a possible world* W *given* E *induced by* $\mathfrak{P}$ *is*

$$\mu_{\mathfrak{P}}(W|E) = \frac{\hat{\mu}_{\mathfrak{P}}(W|E)}{\sum_{W_i \in \Omega(\mathfrak{P})} \hat{\mu}_{\mathfrak{P}}(W_i|E)} = \frac{\hat{\mu}_{\mathfrak{P}}(W|E)}{\sum_{W_i \in \Omega(\mathfrak{P}) \wedge E \subseteq W_i} \hat{\mu}_{\mathfrak{P}}(W_i|E)}$$

When the P-log module $\mathfrak{P}$ *is clear from the context we may simply write* $\hat{\mu}(W|E)$ *and* $\mu(W|E)$ *instead of* $\hat{\mu}_{\mathfrak{P}}(W|E)$ *and* $\mu_{\mathfrak{P}}(W|E)$ *respectively.*

The important remark regarding the above definition is that a possible world W of the P-log module can correspond to several models (the answer sets M_i) of the underlying answer set program, since hidden atoms have been projected out. This way, we need to sum the associated unconditional measures of the ASP models which originate (or contribute to) W. The attentive reader should have noticed that for any world W the unconditional probability measure $\hat{\mu}_{\mathfrak{P}}(W|E)$, for any $E \subseteq W \cap (Rin \cup Rout)$ is identical to $\hat{\mu}_{\mathfrak{P}}(W|W \cap (Rin \cup Rout))$, and zero elsewhere. So, in practice each world just requires one real value to obtain all the conditional probability measures.

Example 4. Construct P-log module $\mathfrak{Sprinkler}$ from Example 1 whose input atoms are $\{cloudy = t, cloudy = f\}$ and output atoms are $\{sprinkler = t, sprinkler = f\}$. The P-log program of $\mathfrak{Sprinkler}$ (with the observation removed) is

$$\begin{array}{l} boolean = \{t, f\}. \qquad cloudy : boolean. \qquad sprinkler : boolean. \\ [rs]\; random(sprinkler, \{X : boolean(X)\}). \\ pr_{rs}(sprinkler = t \mid_c cloudy = f) = 0.5. \\ pr_{rs}(sprinkler = t \mid_c cloudy = t) = 0.1. \end{array}$$

For which, the corresponding ASP program in module $\mathbb{P}(\mathfrak{Sprinkler})$ is:

$hasval(cloudy) :\!-\; cloudy(t).$ $\quad hasval(cloudy) :\!-\; cloudy(f).$
$:\!-$ not $hasval(cloudy).$ $\quad :\!-\; cloudy(t), cloudy(f).$

$-sprinkler(Y1) :\!-\; sprinkler(Y2), Y1! = Y2, boolean(Y1), boolean(Y2).$

$1\{sprinkler(Z) : poss(rsk, sprinkler, Z)\}1 :\!-$ not $intervene(sprinkler).$
$poss(rsk, sprinkler, Z) :\!-\; boolean(Z),$ not $intervene(sprinkler).$
$intervene(sprinkler) :\!-\; do(sprinkler(Y)), boolean(Y).$

$pa(rsk, sprinkler(t), 0.1) :\!-\; poss(rsk, sprinkler, t), cloudy(t).$
$pa(rsk, sprinkler(t), 0.5) :\!-\; poss(rsk, sprinkler, t), cloudy(f).$

$:\!-\; obs(sprinkler(t)),$ not $sprinkler(t).$ $\quad :\!-\; obs(sprinkler(f)),$ not $sprinkler(f).$

Module $\mathbb{P}(\mathfrak{Sprinkler})$ has four answer sets all containing both $poss(rsk, sprinkler, t)$ and $poss(rsk, sprinkler, f)$, and additionally:

$$\begin{aligned} M_1 &= \{sprinkler(t),\ \ cloudy(t),\ \ pa(rsk, sprinkler(t), 0.1)\} \\ M_2 &= \{sprinkler(f),\ \ cloudy(t),\ \ pa(rsk, sprinkler(t), 0.1)\} \\ M_3 &= \{sprinkler(t),\ \ cloudy(f),\ \ pa(rsk, sprinkler(t), 0.5)\} \\ M_4 &= \{sprinkler(f),\ \ cloudy(f),\ \ pa(rsk, sprinkler(t), 0.5)\} \end{aligned}$$

The first two correspond to possible worlds $W_1 = \{sprinkler(t),\ \ cloudy(t)\}$ and $W_2 = \{sprinkler(f),\ \ cloudy(t)\}$ where $cloudy = t$. So, $\hat{\mu}(W_1|\{cloudy(t)\}) = 0.1$ and $\hat{\mu}(W_2|\{cloudy(t)\}) = 0.9$ and $\hat{\mu}(W_3|\{cloudy(t)\}) = \hat{\mu}(W_4|\{cloudy(t)\}) = 0.0$. Since the sum of the unconditional probability measures for all world totals 1.0, then the normalized measure coincides with the unnormalized one for the particular evidence $\{cloudy = t\}$.

Definition 9 (Conditional Probability). *Suppose $\mathfrak{P}$ is a P-log module and $E \subseteq Rin \cup Rout$ for which $\mu_\Pi(.|E)$ is determined. The probability, $P_\mathfrak{P}(A|E)$, of a formula A over literals in Rout, is the sum of the conditional probability measures of the possible worlds of $\mathfrak{P}$ on which A is true, i.e. $P_\mathfrak{P}(A|E) = \sum_{W \vdash A} \mu_\mathfrak{P}(W|E)$.*

The following theorem shows that P-log modules generalize appropriately the notion of conditional probability of P-log programs.

Theorem 2. *Let Π be P-log program Π. Consider the P-log module $\mathfrak{P} = \langle \Pi, \{\}, Lit(\Sigma) \rangle$ then for any set $B \subseteq Lit(\Sigma)$ such that $P_\Pi(B) \neq 0$ then $P_\Pi(A|B) = P_\mathfrak{P}(A|B)$.*

A P-log module corresponds to the notion of factor introduced by [20] in their variable elimination algorithm. The difference is that P-log modules are defined declaratively by a logic program with associated probabilistic semantics, instead of just matrix of values for each possible combination of parameter variables.

4 P-Log Module Theorem

This section provides a way of composing P-log modules and presents the corresponding module theorem. The composition of a P-log module mimics syntactically the composition of an answer set programming module, with similar pre-conditions:

Definition 10 (P-log module composition). *Consider P-log modules $\mathfrak{P}_1 = \langle \Pi_1, Rin_1, Rout_1 \rangle$ over signature Σ_1, and $\mathfrak{P}_2 = \langle \Pi_2, Rin_2, Rout_2 \rangle$ over signature Σ_2, such that:*

1. *$Rout_1 \cap Rout_2 = \emptyset$*
2. *$(Lit(\Sigma_1) \setminus (Rin_1 \cup Rout_1)) \cap Lit(\Sigma_2) = Lit(\Sigma_1) \cap (Lit(\Sigma_2) \setminus (Rin_2 \cup Rout_2)) = \emptyset$*
3. *The sorts of Σ_1 and Σ_2 coincide and are defined equivalently in Π_1 and Π_2.*

The composition of $\mathfrak{P}_1$ *with* $\mathfrak{P}_2$ *is the P-log module* $\mathfrak{P}_1 \oplus \mathfrak{P}_2 = \langle \Pi_1 \cup \Pi_2, (Rin_1 \cup Rin_2) \setminus (Rout_1 \cup Rout_2), (Rout_1 \cup Rout_2)\rangle$ *over signature* $\Sigma_1 \cup \Sigma_2$*.*

The join $\mathfrak{P}_1 \sqcup \mathfrak{P}_2 = \mathfrak{P}_1 \oplus \mathfrak{P}_2$ *is defined in this case whenever additionally there are no dependencies (positive or negative) among modules.*

The first condition forbids the composition of modules having a common output literal, while the second one forbids common hidden atoms (except possibly the sort predicate instances). We avoid joining modules having both negative and positive dependencies.

The compositionality result for P-log modules is more intricate since besides the compositional construction of possible worlds, it is also necessary to ensure compositionality for the underlying conditional probability measures induced by the joined module:

Theorem 3 (P-log Module Theorem). *Consider two P-log modules* $\mathfrak{P}_1$ *and* $\mathfrak{P}_2$ *such that their join* $\mathfrak{P}_1 \sqcup \mathfrak{P}_2$ *is defined. Then* $\Omega(\mathfrak{P}_1 \sqcup \mathfrak{P}_2) = \Omega(\mathfrak{P}_1) \bowtie \Omega(\mathfrak{P}_2)$ *with* $\Omega(\mathfrak{P}_1) \bowtie \Omega(\mathfrak{P}_2) = \{W_1 \cup W_2 \mid W_1 \in \Omega(\mathfrak{P}_1), W_2 \in \Omega(\mathfrak{P}_1),$ *and* $W_1 \cap (Rin_2 \cup Rout_2) = W_2 \cap (Rin_1 \cup Rout_1)\}$*.*

Let $E = E_1 \cup E_2$ *where* $E_1 = E \cap (Rin_1 \cup Rout_1)$ *and* $E_2 = E \cap (Rin_2 \cup Rout_2)$*. Then,* $\hat{\mu}_{\mathfrak{P}_1 \sqcup \mathfrak{P}_2}(W|E) = \hat{\mu}_{\mathfrak{P}_1}(W_1|E_1) \times \hat{\mu}_{\mathfrak{P}_2}(W_2|E_2)$ *with* $W = W_1 \cup W_2$ *such that* $W \in \Omega(\mathfrak{P}_1 \sqcup \mathfrak{P}_2)$*,* $W_1 \in \Omega(\mathfrak{P}_1)$ *and* $W_2 \in \Omega(\mathfrak{P}_2)$*.*

Notice that the P-log module theorem is defined only in terms of the unnormalized conditional probability measures. The normalized ones can be obtained as in the previous case dividing by the sum of unconditional measure of all worlds given the evidence. Again, we just have to consider one value for each world (i.e. when evidence is maximal).

The application to Bayesian Networks is now straightforward. First, each random variable in a Bayesian Network is captured by a P-log module having the corresponding attribute literals of the random variable as output literals, and input literals are all attribute literals obtainable from parent variables. The conditional probability tables are represented by pr-atoms, as illustrated before in Example 1. P-log module composition inherits associativity and commutativity from ASP modules, and thus P-log modules can be joined in arbitrary ways since there are no common output atoms, and there are no cyclic dependencies.

The important remark is that a P-log module is an extension of the notion of factor used in the variable elimination algorithm [20]. We only need a way to eliminate variables from a P-log module in order to simulate the behaviour of the variable elimination algorithm, but this is almost trivial:

Definition 11 (Eliminate operation). *Consider a P-log module* $\mathfrak{P} = \langle \Pi, Rin, Rout\rangle$ *over signature* Σ*, and a subset of attribute literals* $S \subseteq Rout$*. Then, P-log module* $Elim(\mathfrak{P}, S) = \langle \Pi, Rin, Rout \setminus S\rangle$ *eliminates (hides) from* $\mathfrak{P}$ *the attribute literals in* S*.*

By hiding all attribute literals of a given random variable, we remove the random attribute from the possible worlds (as expected), summing away corresponding original possible worlds. By applying the composition of P-log modules and eliminate operations by the order they are performed by the variable elimination algorithm, the exact behaviour of the variable elimination algorithm is attained. Thus, for the case of polytrees represented in P-log, we can do reasoning with P-log in polynomial time.

5 Conclusions and Future Work

We present the first approach in the literature to modularize P-log programs and to make their composition incrementally by combining compatible possible worlds and multiplying corresponding unnormalized conditional probability measures. A P-log module corresponds to a factor of the variable elimination algorithm [20,18], clarifying and improving the relationship of P-log with traditional Bayesian Network approaches. By eliminating variables in P-log modules we may reduce the space and time necessary to make inference in P-log, in contrast with previous algorithms [1,8] which require always enumeration of the full possible worlds (which are exponential on the number of random variables) and repeat calculations. As expected, it turns out that the general case of exact inference is intractable, so we must consider methods for approximate inference.

As future work, we intend to fully describe the inference algorithm obtained from the compositional semantics of P-log modules and relate it formally with the variable elimination algorithm. Furthermore we expect that the notion of P-log module may also help to devise approximate inference methods, e.g. by extending sampling algorithms, enlarging the applicability of P-log which is currently somehow restricted. Finally, we hope to generalize the P-log language to consider other forms of uncertainty representation like belief functions, possibility measures or even plausibility measures [9].

References

1. Anh, H.T., Kencana Ramli, C.D.P., Damásio, C.V.: An implementation of extended P-log using XASP. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 739–743. Springer, Heidelberg (2008)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
3. Baral, C., Gelfond, M., Rushton, J.N.: Probabilistic reasoning with answer sets. TPLP 9(1), 57–144 (2009)
4. Baral, C., Hunsaker, M.: Using the probabilistic logic programming language P-log for causal and counterfactual reasoning and non-naive conditioning. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence, pp. 243–249. Morgan Kaufmann Publishers Inc., San Francisco (2007)
5. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular Nonmonotonic Logic Programming Revisited. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 145–159. Springer, Heidelberg (2009)
6. Gaifman, H., Shapiro, E.: Fully abstract compositional semantics for logic programs. In: POPL 1989: Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 134–142. ACM, New York (1989)

7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) Proceedings of the Fifth International Conference and Symposium (ICLP/SLP), pp. 1070–1080. MIT Press, Cambridge (1988)
8. Gelfond, M., Rushton, N., Zhu, W.: Combining logical and probabilistic reasoning. In: Proc. of AAAI 2006 Spring Symposium: Formalizing AND Compiling Background Knowledge AND Its Applications to Knowledge Representation AND Question Answering, pp. 50–55. AAAI Press, Menlo Park (2006)
9. Halpern, J.Y.: Reasoning about Uncertainty. The MIT Press, Cambridge (2005)
10. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Kemper, P. (ed.) Proc. Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems, pp. 7–12 (September 2001)
11. Lifschitz, V.: Twelve definitions of a stable model. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 37–51. Springer, Heidelberg (2008)
12. Lifschitz, V.: What is answer set programming? In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 1594–1597. MIT Press, Cambridge (2008)
13. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. Theory Pract. Log. Program. 8(5-6) (2008)
14. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers Inc., San Francisco (1988)
15. Pearl, J.: Causality: Models, Reasoning and Inference. Cambridge Univ. Press, Cambridge (2000)
16. Pfeffer, A., Koller, D.: Semantics and inference for recursive probability models. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, pp. 538–544. AAAI Press, Menlo Park (2000)
17. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artif. Intell. 94, 7–56 (1997)
18. Poole, D., Zhang, N.L.: Exploiting contextual independence in probabilistic inference. J. Artif. Int. Res. 18, 263–313 (2003)
19. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall, Englewood Cliffs (2010)
20. Zhang, N., Poole, D.: A simple approach to Bayesian network computations. In: Proceedings of the Tenth Canadian Conference on Artificial Intelligence, pp. 171–178 (1994)

Symmetry Breaking for Distributed Multi-Context Systems*

Christian Drescher[1], Thomas Eiter[2], Michael Fink[2], Thomas Krennwallner[2], and Toby Walsh[1]

[1] NICTA and University of New South Wales
Locked Bag 6016, Sydney NSW 1466, Australia
{christian.drescher,toby.walsh}@nicta.com.au
[2] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink,tkren}@kr.tuwien.ac.at

Abstract. Heterogeneous nonmonotonic multi-context systems (MCS) permit different logics to be used in different contexts, and link them via bridge rules. We investigate the role of symmetry detection and symmetry breaking in such systems to eliminate symmetric parts of the search space and, thereby, simplify the evaluation process. We propose a distributed algorithm that takes a local stance, i.e., computes independently the partial symmetries of a context and, in order to construct potential symmetries of the whole, combines them with those partial symmetries returned by neighbouring contexts. We prove the correctness of our methods. We instantiate such symmetry detection and symmetry breaking in a multi-context system with contexts that use answer set programs, and demonstrate computational benefit on some recently proposed benchmarks.

1 Introduction

Due to the increasing application of distributed systems, there has been recent interest in formalisms that accommodate several, distributed knowledge bases. Based on work by McCarthy [14] and Giunchiglia [11], a powerful approach is multi-context systems (MCS; [12]). Intuitively, an MCS consists of several heterogeneous theories (the contexts), which may use different logical languages and different inference systems, that are interlinked with a special type of rules that allow to add knowledge into a context depending on knowledge in other contexts. MCSs have applications in various areas such as argumentation, data integration, and multi-agent systems. In the latter, each context models the beliefs of an agent while the bridge rules model an agent's perception of the environment. Among various proposals for MCS, the general MCS framework of Brewka and Eiter [5] is of special interest, as it generalises previous approaches in contextual reasoning and allows for heterogeneous and nonmonotonic MCSs. Such a

* This research has been supported by the Austrian Science Fund project P20841 and by the Vienna Science and Technology Fund project ICT 08-020. NICTA is funded by the Department of Broadband, Communications and the Digital Economy, and the Australian Research Council.

system can have different, possibly nonmonotonic logics in the different contexts, e.g., answer set programs (ASP; [4]), and bridge rules can use default negation to deal with incomplete information.

Although there has been dramatic improvements [3] in the performance of distributed algorithms for evaluating Brewka and Eiter' style nonmonotonic MCSs such as DMCS [7], many applications exhibit symmetries. For example, suppose context C_1 is an advanced database system which repairs inconsistencies (e.g., from key violations in database tables), and another context C_2 is accessing the repaired tables via bridge rules. A large (exponential) number of repairs may exist, each yielding a local model (i.e., belief set) of C_1; many of those models are symmetric, thus C_2's bridge rules may fire for many symmetric repairs. This can frustrate an evaluation algorithm as it fruitlessly explores symmetric subspaces. Furthermore, communicating symmetric solutions from one context to another can impede further search. If symmetries can be identified, we can avoid redundant computation by pruning parts of the search space through symmetry breaking. However, symmetry breaking in MCSs has not been explored in any depth.

In order to deal with symmetry in MCSs, we must accomplish two tasks: (1) identifying symmetries and (2) breaking the identified symmetries. We make several fundamental and foundational contributions to the study of symmetry in MCS.

– First, we define the notion of symmetry for MCSs. This is subsequently specialized to local symmetries and partial symmetries that capture symmetry on parts of an MCS. Partial symmetries can be extended to a symmetry of the whole system under suitable conditions which are formalized in a corresponding notion of join.
– Second, we design a distributed algorithm to identify symmetries based on such partial symmetries. The method runs as background processes in the contexts and communicate with each other for exchanging partial symmetries. This algorithm computes symmetries of a general MCS based on the partial symmetries for each individual context. We demonstrate such symmetry detection for ASP contexts using automorphisms of a suitable coloured graph.
– Third, we break symmetries by extending the symmetry breaking methods of Crawford *et al.* [6] to distributed MCS. We construct symmetry-breaking constraints (SBCs) for a MCS that take into account beliefs imported from other contexts into account. These constraints ensure that an evaluation engine never visits two points in the search space that are symmetric. For contexts other than propositional logic, distributed SBCs have to be expressed appropriately. Again we illustrate this in the case of ASP contexts and develop a logic-program encoding for distributed symmetry breaking constraints.
– Finally, we experimentally evaluate our approach on MCSs with ASP contexts. In problems with large number of symmetries, we demonstrate the effectiveness of only breaking a subset of the symmetries. Results on MCS benchmarks that resemble context dependencies of realistic scenarios [3] show that symmetry breaking yields significant improvements in runtime and compression of the solution space.

2 Logical Background

We recall some basic notions of heterogeneous nonmonotonic multi-context systems. Following [5], a *logic* over an alphabet $\mathcal{A}$ is a triple $L = (\mathrm{KB}, \mathrm{BS}, \mathrm{ACC})$, where KB is a set of well-formed knowledge bases over $\mathcal{A}$, BS is a set of possible belief sets (sets over $\mathcal{A}$), and ACC: $\mathrm{KB} \rightarrow 2^{\mathrm{BS}}$ is a function describing the semantics of the logic by assigning each $kb \in \mathrm{KB}$ a set of acceptable sets of beliefs. This covers many monotonic and nonmonotonic logics like *propositional logic* under the closed world assumption and *default logic*. We concentrate on logic programs under answer set semantics, i.e., ASP logic L. A (disjunctive) *logic program* over an alphabet $\mathcal{A}$ is a finite set of rules

$$a_1; \ldots ; a_\ell \leftarrow b_1, \ldots, b_j, {\sim}b_{j+1}, \ldots, {\sim}b_m \tag{1}$$

where $a_i, b_k \in \mathcal{A}$ for $1 \leq i \leq \ell$, and $1 \leq k \leq m$. A *literal* is an atom a or its default negation ${\sim}a$. For a rule r, let $\mathrm{head}(r) = \{a_1, \ldots, a_\ell\}$ be the *head* of r and $\mathrm{body}(r) = \{b_1, \ldots, b_j, {\sim}b_{j+1}, \ldots, {\sim}b_m\}$ the *body* of r. For an ASP logic L, the set of knowledge bases KB is given through the set of logic programs, the possible belief sets $\mathrm{BS} = 2^{\mathcal{A}}$ contains all subsets of atoms, and $\mathrm{ACC}(P)$ is the set of answer sets of a logic program P. For a detailed introduction to ASP, we refer to [4].

We now recall multi-context systems according to Brewka and Eiter [5]. A *multi-context system* $M = (C_1, \ldots, C_n)$ consists of a collection of contexts $C_i = (L_i, kb_i, br_i)$, where $L_i = (\mathrm{KB}_i, \mathrm{BS}_i, \mathrm{ACC}_i)$ is a logic over alphabets $\mathcal{A}_i$, $kb_i \in \mathrm{KB}_i$ is a knowledge base, and br_i is a set of L_i *bridge rules* r of the form

$$a \leftarrow (c_1 : b_1), \ldots, (c_j : b_j), {\sim}(c_{j+1} : b_{j+1}), \ldots, {\sim}(c_m : b_m) \ , \tag{2}$$

where $1 \leq c_k \leq n$, b_k is an atom in $\mathcal{A}_{c_k}$, $1 \leq k \leq m$, and $kb \cup \{a\} \in \mathrm{KB}_i$ for each $kb \in \mathrm{KB}_i$. We call a *context atom* $(c_k : b_k)$ or its default negation ${\sim}(c_k : b_k)$ a *context literal*. Analogous to standard notions of ASP, let the atom $\mathrm{head}(r) = a$ be the *head* of r and $\mathrm{body}(r) = \{(c_1 : b_1), \ldots, (c_j : b_j), {\sim}(c_{j+1} : b_{j+1}), \ldots, {\sim}(c_m : b_m)\}$ the *body* of r. For a set S of context literals, define $S^+ = \{(c : b) \mid (c : b) \in S\}$, $S^- = \{(c : b) \mid {\sim}(c : b) \in S\}$, and for a set S of context atoms, let $S|_c = \{b \mid (c : b) \in S\}$. The set of atoms occurring in a set br_i of bridge rules is denoted by $\mathrm{at}(br_i)$. W.l.o.g., we will assume that the alphabets $\mathcal{A}_i$ are pairwise disjoint and denote their union by $\mathcal{A} = \bigcup_{i=1}^{n} \mathcal{A}_i$.

Intuitively, context literals in bridge rules refer to information of other contexts. Bridge rules can thus modify the knowledge base, depending on what is believed or disbelieved in other contexts. The semantics of an MCS is given by its equilibria, which is a collection of acceptable belief sets, one from each context, that respect all bridge rules. More formally, for an MCS $M = (C_1, \ldots, C_n)$ define a *belief state* $S = (S_1, \ldots, S_n)$ of M such that each $S_i \in \mathrm{BS}_i$. A bridge rule r of the form (2) is *applicable* in S iff $\mathrm{body}(r)^+|_{c_k} \subseteq S_{c_k}$ and $\mathrm{body}(r)^-|_{c_k} \cap S_{c_k} = \emptyset$ for all $1 \leq k \leq m$. A belief state $S = (S_1, \ldots, S_n)$ of an MCS $M = (C_1, \ldots, C_n)$ is an *equilibrium* iff $S_i \in \mathrm{ACC}_i(kb_i \cup \{\mathrm{head}(r) \mid r \in br_i,\ r \text{ is applicable in } S\})$ for all $1 \leq i \leq n$.

In practice, however, we are more interested in equilibria of a *subsystem* with root context C_k, e.g., when querying to a context. Naturally, such partial equilibria have to contain coherent information from C_k and all contexts in the import closure of C_k, and therefore, are parts of potential equilibria of the whole system. We define the *import neighbourhood* of a context C_k as the set $In(k) = \{c \mid (c : b) \in \text{body}(r), r \in br_k\}$ and the *import closure* $IC(k)$ as the smallest set of contexts S such that (1) $C_k \in S$ and (2) $C_i \in S$ implies $\{C_j \mid j \in In(i)\} \subseteq S$. Let $\varepsilon \notin \mathcal{A}$ be a new symbol representing the value 'unknown'. A *partial belief state* of M is a sequence $S = (S_1, \dots, S_n)$, such that $S_i \in \text{BS}_i \cup \{\varepsilon\}$ for all $1 \leq i \leq n$. A partial belief state $S = (S_1, \dots, S_n)$ of MCS $M = (C_1, \dots, C_n)$ w.r.t. C_k is a *partial equilibrium* iff whenever $C_i \in IC(k)$, $S_i \in \text{ACC}_i(kb_i \cup \{\text{head}(r) \mid r \in br_i,\ r \text{ is applicable in } S\})$, otherwise $S_i = \varepsilon$, for all $1 \leq i \leq n$.

Example 1. As a running example, consider the MCS $M = (C_1, C_2, C_3)$ with ASP logics over alphabets $\mathcal{A}_1 = \{a, b, c\}$, $\mathcal{A}_2 = \{d, e, f, g\}$, and $\mathcal{A}_3 = \{h\}$. Suppose

$$kb_1 = \left\{ c \leftarrow a, b, {\sim}c \right\}, \quad kb_2 = \left\{ \begin{array}{l} f \leftarrow d, e, {\sim}g \\ g \leftarrow d, e, {\sim}f \end{array} \right\}, \quad kb_3 = \emptyset,$$

$$br_1 = \left\{ \begin{array}{l} a \leftarrow {\sim}(2 : d) \\ b \leftarrow {\sim}(2 : e) \end{array} \right\}, \quad br_2 = \left\{ \begin{array}{l} d \leftarrow {\sim}(1 : a) \\ e \leftarrow {\sim}(1 : b) \end{array} \right\}, \quad br_3 = \left\{ h \leftarrow (1 : a) \right\}.$$

Then, $(\{b\}, \{d\}, \varepsilon)$, $(\{a\}, \{e\}, \varepsilon)$, $(\emptyset, \{d, e, f\}, \varepsilon)$, and $(\emptyset, \{d, e, g\}, \varepsilon)$ are partial equilibria w.r.t. C_1, and $(\{b\}, \{d\}, \emptyset)$, $(\{a\}, \{e\}, \{h\})$, $(\emptyset, \{d, e, f\}, \emptyset)$, and $(\emptyset, \{d, e, g\}, \emptyset)$ are equilibria. Observe that M remains invariant under a swap of atoms f and g, which is what we will call a symmetry of M. Furthermore, the subsystem given by $IC(1) = \{C_1, C_2\}$ remains invariant under a swap of atoms f and g, and/or a simultaneous swap of atoms a, b and d, e, which is what we will call a partial symmetry of M w.r.t. $\{C_1, C_2\}$.

3 Algebraic Background

Intuitively, a symmetry of a discrete object is a transformation of its components that leaves the object unchanged. Symmetries are studied in terms of groups. Recall that a *group* is an abstract algebraic structure $(G, *)$, where G is a set closed under a binary associative operation $*$ such that there is a *unit* element and every element has a unique *inverse*. Often, we abuse notation and refer to the group G, rather than to the structure $(G, *)$, and we denote the size of G as $|G|$. A compact representation of a group is given through generators. A set of group elements such that any other group element can be expressed in terms of their product is called a *generating set* or *set of generators*, and its elements are called *generators*. A generator is *redundant*, if it can be expressed in terms of other generators. A generating set is *irredundant*, if no strict subset of it is generating. Such a set provides an extremely compact representation of a group. In fact, representing a finite group by an irredundant generating set ensures exponential compression, as it contains at most $\log_2 |G|$ elements [1].

A mapping $f\colon G \to H$ between two groups $(G, *)$ and $(H, \circ)$ is a *homomorphism* iff for $a, b \in G$ we have that $f(a * b) = f(a) \circ f(b)$; if it has also an inverse that is a homomorphism, f is an *isomorphism*, which is an *automorphism* if $G = H$. The groups G and H are called *isomorphic*, if there exists some isomorphism between them. Any group isomorphism maps (irredundant) generating sets to (irredundant) generating sets [1]. The domain G of f is denoted as $\mathrm{dom}(f)$. In our context, the group of permutations is most important. Recall that a *permutation* of a set S is a bijection $\pi\colon S \rightarrow S$. It is well-known that the set of all permutations of S form a group under composition, denoted as $\Pi(S)$.

The image of $a \in S$ under a permutation π is denoted as a^π, and for vectors $s = (a_1, a_2, \ldots, a_k) \in S^k$ define $s^\pi = (a_1^\pi, a_2^\pi, \ldots, a_k^\pi)$. For formulas $\phi(a_1, a_2, \ldots, a_k)$ of some logic over alphabet $\mathcal{A}$ s. t. $S \subseteq \mathcal{A}$ define $\phi^\pi(a_1, a_2, \ldots, a_k) = \phi(a_1^\pi, a_2^\pi, \ldots, a_k^\pi)$, e.g., for a rule r of form (1), let r^π be $a_1^\pi; \ldots ; a_\ell^\pi \leftarrow b_1^\pi, \ldots, b_j^\pi, {\sim}b_{j+1}^\pi, \ldots, {\sim}b_m^\pi$. For a bridge rule r of form (2) define $r^\pi = a^\pi \leftarrow (c_1 : b_1^\pi), \ldots, (c_j : b_j^\pi), {\sim}(c_{j+1} : b_{j+1}^\pi), \ldots, {\sim}(c_m : b_m^\pi)$. Finally, for a set X (of elements or subsets from S, formulas, bridge rules, etc.), define $X^\pi = \{x^\pi \mid x \in X\}$.

We will make use of the *cycle notation* where a permutation is a product of disjoint cycles. A cycle $(a_1\ a_2\ a_3\ \cdots\ a_n)$ means that the permutation maps a_1 to a_2, a_2 to a_3, and so on, finally a_n back to a_1. An element that does not appear in any cycle is understood as being mapped to itself. The *orbit* of $a \in S$ under a permutation $\pi \in \Pi(S)$ are the set of elements of S to which a can be mapped by (repeatedly) applying π. Note that orbits define an equivalence relation on elements (sets, vectors, etc.) of S.

In graph theory, the symmetries are studied in terms of graph automorphisms. We consider directed graphs $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges. Intuitively, an automorphism of G is a permutation of its vertices that maps edges to edges, and non-edges to non-edges, preserving edge orientation. More formally, an *automorphism* or a *symmetry of* G is a permutation $\pi \in \Pi(V)$ such that $(u, v)^\pi \in E$ iff $(u, v) \in E$. An extension considers vertex colourings that are partitionings $\rho(V) = \{V_1, V_2, \ldots, V_k\}$ of the nodes V into disjoint nonempty sets ("colours") V_i. Symmetries must map each vertex to a vertex with the same colour. Formally, given a colouring of the vertices $\rho(V) = \{V_1, V_2, \ldots, V_k\}$, an *automorphism* or a *symmetry of a coloured graph* G is a symmetry π of G s.t. $\rho(V)^\pi = \rho(V)$. The *graph automorphism* problem (GAP) is to find all symmetries of a given graph, for instance, in terms of generators. GAP is not known to be solvable in polynomial time, and its decisional variant is known to be within the complexity classes P and NP, but there is strong evidence that this problem is not NP-complete (cf. [2]). Thus it is potentially easier than, for instance, deciding answer set existence.

4 Symmetry in Multi-Context Systems

We will now define our notion of a symmetry of a multi-context system. In this section we consider MCS $M = (C_1, \ldots, C_n)$ with logics L_i over alphabet $\mathcal{A}_i$, for $1 \leq i \leq n$.

Definition 1. *A* symmetry of M *is a permutation* $\pi \in \Pi(\mathcal{A})$ *such that (1)* $\mathcal{A}_i^\pi = \mathcal{A}_i$*, (2)* $kb_i^\pi = kb_i$*, and (3)* $br_i^\pi = br_i$*, for* $1 \leq i \leq n$*.*

In this definition, items (2) and (3) capture the intention that symmetries are permutations of beliefs which yield identical knowledge bases and bridge rules, respectively. Item (1) imposes that symmetries do not alter the indiviuale context languages; there is no technical need for this, i.e., dropping (1) would yield a more general definition of symmetry for which our subsequent results would still hold; however the respective additional symmetries are irrelevant from a practical point of view and thus disregarded. For the same reason, we disregard permutations of the order of contexts.

Sometimes, a symmetry affects only atoms of a single context, i.e., behaves like the identity for the atoms of all other contexts. A symmetry π of M is *local* for context C_k iff $a^\pi = a$ for all $a \in \text{dom}(\pi) \setminus \mathcal{A}_k$.

Example 2 (cont'd). Reconsider the MCS $M = (C_1, C_2, C_3)$ from Example 1. Symmetries of M are given through the identity and $(f\ g)$, both are local for C_2.

Similar to belief states, we define the notion of partial symmetries, which are parts of potential symmetries of the system.

Definition 2. *A permutation π of the elements in $S \subseteq \mathcal{A}$ is a* partial symmetry *of M w.r.t. the set of contexts $C = \{C_{i_1}, \dots, C_{i_m}\}$ iff (1) $\mathcal{A}_{i_k} \cup at(br_{i_k}) \subseteq S$ (2) $\mathcal{A}_{i_k}^\pi = \mathcal{A}_{i_k}$, (3) $kb_{i_k}^\pi = kb_{i_k}$, and (4) $br_{i_k}^\pi = br_{i_k}$, for all $1 \leq k \leq m$.*

For combining partial symmetries π and σ, we define their *join* $\pi \bowtie \sigma$ as the permutation θ, where

$$a^\theta = \begin{cases} a^\pi & \text{if } a \in \text{dom}(\pi), \\ a^\sigma & \text{if } a \in \text{dom}(\sigma). \end{cases}$$

whenever $a^\pi = a^\sigma$ for all $a \in \text{dom}(\pi) \cap \text{dom}(\sigma)$; otherwise, the join is undefined. The *join* of two sets of partial symmetries of M is naturally defined as $\Pi \bowtie \Sigma = \{\pi \bowtie \sigma \mid \pi \in \Pi,\ \sigma \in \Sigma\}$. Note that, $\pi \bowtie \sigma$ is void, i.e., undefined, if π and σ behave different for some $a \in \text{dom}(\pi) \cap \text{dom}(\sigma)$. Otherwise, the join is a partial symmetry of M.

Theorem 1. *Let $M = (C_1, \dots, C_n)$ be an MCS with logics L_i over alphabet $\mathcal{A}_i$. (1) Every partial symmetry of M w.r.t. $\{C_1, \dots, C_n\}$ is a symmetry of M. (2) For every partial symmetries π and σ of M w.r.t. $C_{(\pi)} = \{C_{i_1}, \dots, C_{i_m}\}$ and $C_{(\sigma)} = \{C_{j_1}, \dots, C_{j_\ell}\}$, respectively, such that $\theta = \pi \bowtie \sigma$ is defined, θ is a partial symmetry of M w.r.t. $C_{(\pi)} \cup C_{(\sigma)}$.*

Proof. (1) Let θ be a partial symmetry of M w.r.t. $\{C_1, \dots, C_n\}$. By Definition 2 we have $\text{dom}(\theta) \subseteq \bigcup_{i=1}^n \mathcal{A}_i = \mathcal{A}$ (an upper bound for the domain of partial symmetries), and $\mathcal{A}_i \subseteq \text{dom}(\pi)$ (lower bound for domain of partial symmetries) for $1 \leq i \leq n$. Hence, θ is a permutation of exactly the elements in $\mathcal{A}$. Given this, and since $\mathcal{A}_i^\theta = \mathcal{A}_i$, $kb_i^\theta = kb_i$ and $br_i^\theta = br_i$ holds for $1 \leq i \leq n$, i.e., all contexts in M, we have that θ is a symmetry of M. (2) We check that all conditions of a partial symmetry hold for θ. By definition of the join, $\text{dom}(\theta) = \text{dom}(\pi) \cup \text{dom}(\sigma) \supseteq \bigcup_{k=1}^m (\mathcal{A}_{i_k} \cup \text{at}(br_{i_k})) \cup \bigcup_{k=1}^\ell (\mathcal{A}_{j_k} \cup \text{at}(br_{j_k}))$. Furthermore, $\mathcal{A}_{i_k}^\theta = \mathcal{A}_{i_k}^\pi = \mathcal{A}_{i_k}$, $kb_{i_k}^\theta = kb_{i_k}^\pi = kb_{i_k}$ and $br_{i_k}^\theta = br_{i_k}^\pi = br_{i_k}$ for all $1 \leq k \leq m$, and similarly, $\mathcal{A}_{j_k}^\theta = \mathcal{A}_{j_k}^\sigma = \mathcal{A}_{j_k}$, $kb_{j_k}^\theta = kb_{j_k}^\sigma = kb_{j_k}$ and $br_{j_k}^\theta = br_{j_k}^\sigma = br_{j_k}$ for all $1 \leq k \leq \ell$. Hence, θ is a partial symmetry of M w.r.t. $C_{(\pi)} \cup C_{(\sigma)}$. □

Observe that every partial symmetry of M w.r.t. a set of contexts C is a partial symmetry of M w.r.t. a non-empty subset of C; a partial symmetry can always be written as the join of two partial symmetries.

Example 3 (cont'd). Reconsider M from Example 1. The partial symmetries Π of M w.r.t. $\{C_1\}$ are given through the identity id and $(a\ b)\ (d\ e)$. The partial symmetries Σ of M w.r.t. $\{C_2\}$ are given through id, $(a\ b)\ (d\ e)\ (f\ g)$, and $(f\ g)$. The partial symmetries of M w.r.t. $\{C_1, C_2\}$ are $\Pi \bowtie \Sigma = \Sigma$, and the partial symmetries Θ of M w.r.t. $\{C_3\}$ are just id alone. The symmetries of M are $\Pi \bowtie \Theta = \{\text{id}, (f\ g)\}$.

5 Distributed Symmetry Detection

In the following, we provide a distributed algorithm for detecting symmetries of an MCS $M = (C_1, \ldots, C_n)$. We follow Dao-Tran *et al.* [7] by taking a local stance, i.e., we consider a context C_k and those parts of the system that are in the import closure of C_k to compute (potential) symmetries of the system. To this end, we design an algorithm whose instances run independently at each context node and communicate with other instances for exchanging sets of partial symmetries. This provides a method for distributed symmetry building.

The idea is as follows: starting from a context C_k, we visit the import closure of C_k by expanding the import neighbourhood at each context, maintaining the set of visited contexts in a set H, the *history*, until a leaf context is reached, or a cycle is detected by noticing the presence of a neighbour context in H. A leaf context C_i simply computes all partial symmetries of M w.r.t. $\{C_i\}$. Then, it returns the results to its parent (the invoking context), for instance, in form of permutation cycles. The results of intermediate contexts C_i are partial symmetries of M w.r.t. $\{C_i\}$, which can be joined, i.e., consistently combined, with partial symmetries from their neighbours, and resulting in partial symmetries of M w.r.t. $IC(i)$. In particular, the starting context C_k returns its partial symmetries joined with the results from its neighbours, as a final result. We assume that each context C_k has a background process that waits for incoming requests with history H, upon which it starts the computation outlined in our algorithm shown in Fig. 1. We write C_i.DSD(H) to specify that we send H to the process at context C_i and wait for its return message. This process also serves the purpose of keeping the cache $c(k)$ persistent. We use the primitive LSD(C_k) which computes all partial symmetries of M w.r.t. $\{C_k\}$ over $\mathcal{A}_k \cup \text{at}(br_k)$.

```
Algorithm: DSD$(H)$ at context $C_k$
Input:     Visited contexts $H$.
Data:      Cache $c(k)$.
Output:    The set of accumulated partial symmetries $\Pi$.

  if $c(k)$ is not initialised then $c(k) \leftarrow$ LSD$(C_k)$;
  $H \leftarrow H \cup \{k\}$;
  $\Pi \leftarrow c(k)$;
  foreach $i \in In(k) \setminus H$ do $\Pi \leftarrow \Pi \bowtie C_i$.DSD$(H)$;
  return $\Pi$;
```

Fig. 1. The distributed symmetry detection algorithm

Our algorithm proceeds in the following way:

1. Check the cache for partial symmetries of M w.r.t. $\{C_k\}$;
2. if imports from neighbour contexts are needed, then request partial symmetries from all neighbours and join them (previously visited contexts excluded). This can be performed in parallel. Also, partial symmetries can be joined in the order neighbouring contexts do answer; and
3. return partial symmetries of M w.r.t. $IC(k)$.

Correctness of our approach hold by the following result.

Theorem 2. *Let $M = (C_1, \ldots, C_n)$ be an MCS and C_k be a context in M. Then, $\pi \in C_k$.DSD$(\emptyset)$ iff π is a partial symmetry of M w.r.t. $IC(k)$.*

Proof (sketch). ($\Rightarrow$) We prove soundness, i.e., if $\pi \in C_k$.DSD$(\emptyset)$ then π is a partial symmetry of M w.r.t. $IC(k)$. We proceed by structural induction on the topology of an MCS, and start with acyclic MCS M. Base case: C_k is a leaf with $br_k = \emptyset$ and $In(k) = \emptyset$. By assumption, LSD(C_k) computes all partial symmetries of M w.r.t. $\{C_k\}$, i.e., $c(k) \leftarrow$ LSD(C_k) in the algorithm in Fig. 1. Induction step: for non-leaf C_k, suppose $In(k) = \{i_1, \ldots, i_m\}$ and $\Pi_k =$ LSD(C_k), $\Pi_{i_j} = C_{i_j}$.DSD$(H \cup \{k\})$ for $1 \leq j \leq m$. By Theorem 1, $\Pi = \Pi_k \bowtie \Pi_{i_1} \bowtie \cdots \bowtie \Pi_{i_m}$, as computed by $\Pi \leftarrow \Pi \bowtie C_i$.DSD$(H)$ in the loop of the algorithm in Fig. 1, consists of partial symmetries of M w.r.t. $IC(k)$.

The proof for cyclic M is similar. In a run we eventually end up in C_i such that $i \in H$ again. In that case, calling C_i.DSD(H) is discarded, which breaks the cycle. However, partial symmetries excluding C_i are propagated through the system to the calling C_i which combines the intermediate results with partial symmetries of M w.r.t. $\{C_i\}$.
($\Leftarrow$) We give now a proof sketch for completeness. Let π be a partial symmetry of M w.r.t. $IC(k)$. We show $\pi \in C_k$.DSD$(\emptyset)$. The proof idea is as follows: we proceed as in the soundness part by structural induction on the topology of M, and in the base case for a leaf context C_k, by assumption, we get that LSD(C_k) returns all partial symmetries of M w.r.t. $\{C_k\}$, i.e., all partial symmetries of M w.r.t. $IC(k)$. For the induction step, we verify straightforward that π being a partial symmetry of M w.r.t. $IC(k)$ implies π being a partial symmetry of M w.r.t. $IC(i)$ for all $i \in In(k)$. □

6 Symmetry Detection via Graph Automorphism

The primitive LSD(C_i) for detecting partial symmetries of an MCS $M = (C_1, \ldots, C_n)$ w.r.t. $\{C_i\}$ using logic L_i has to be defined for every logic L_i anew. As an example, our approach for detecting partial symmetries of M w.r.t. an ASP context C_i is through reduction to, and solution of, an associated graph automorphism problem. The graph $GAP(C_i)$ is constructed as follows:

1. Every atom that occurs in $kb_i \cup br_i$ (every context atom $(c : b)$ in br_i, respectively) is represented by two vertices of colour i (c, respectively) and $n+1$ that correspond to the positive and negative literals.
2. Every rule (every bridge rule, respectively) is represented by a *body vertex* of colour $n + 2$ ($n + 3$, respectively), a set of directed edges that connect the vertices of the literals (context literals, respectively) that appear in the rule's body to

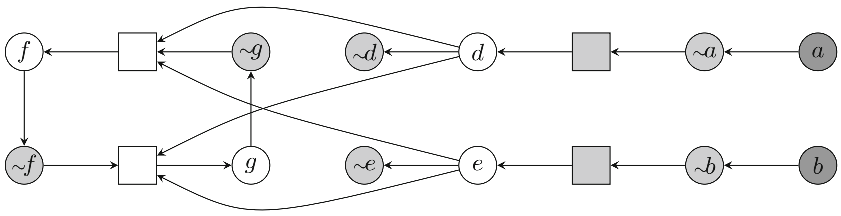


Fig. 2. GAP reduction of context C_2 from Example 1

its body vertex, and a set of directed edges that connect the body vertex to the vertices of the atoms that appear in the head of the rule.

3. To properly respect negation, that is, an atom a maps to b if and only if $\sim a$ maps to $\sim b$ for any atoms a and b, vertices of opposite (context) literals are mated by a directed edge from the positive (context) literal to the negative (context) literal.

Example 4 (cont'd). Reconsider MCS M from Example 1. Fig. 2 illustrates $GAP(C_2)$, where different shapes and tones represent different colours.

Symmetries of $GAP(C_i)$ correspond precisely to the partial symmetries of M w.r.t. $\{C_i\}$.

Theorem 3. *Let $M = (C_1, \ldots, C_n)$ be an MCS with ASP context C_i. The partial symmetries of M w.r.t. $\{C_i\}$ correspond one-to-one to the symmetries of $GAP(C_i)$.*

Proof. The proof for logic programs is shown in [8]. Therefore we only provide arguments regarding bridge rules and context atoms. ($\Rightarrow$) A partial symmetry of M w.r.t. $\{C_i\}$ will map context atoms to context atoms of the same context. Since they have the same colour, the symmetry is preserved for corresponding vertices and consistency edges. The same applies to body vertices and edges representing bridge rules, since the body vertices have incoming edges from context literal vertices with their respective colour only, and vertices of the same colour are mapped one to another. Thus, a consistent mapping of atoms in C_k, when carried over to the graph, must preserve symmetry. ($\Leftarrow$) We now show that every symmetry in the graph corresponds to a partial symmetries of M w.r.t. $\{C_i\}$. Recall that we use one colour for positive context literals from each context, one for negative context literals from each context, and one for bodies. Hence, a graph symmetry must map (1) positive context literal vertices to other such from the same context, negative literal vertices to negative literal vertices from the same context, and body vertices to body vertices, and (2) the body edges of a vertex to body edges of its mate. This is consistent with partial symmetries of M w.r.t. $\{C_i\}$ mapping context atoms to context atoms, and bodies to bodies, i.e., bridge rules to bridge rules. $\square$

Corollary 1. *Let $M = (C_1, \ldots, C_n)$ be an MCS with ASP context C_i. The partial symmetry group of M w.r.t. $\{C_i\}$ and the symmetry group of $GAP(C_i)$ are isomorphic. Furthermore, sets of partial symmetry generators of M w.r.t. $\{C_i\}$ correspond one-to-one to sets of symmetry generators of $GAP(C_i)$.*

To detect local symmetries only, we further modify our approach by assigning a unique colour to each context atom and each atom that is referenced in other contexts, i.e., context atoms cannot be mapped.

With reference to related work (cf. [1,8]), we stretch that the detection of symmetries through reduction to graph automorphism is computationally quite feasible, i.e., the overhead cost in situations that do not have symmetries is negligible.

7 Distributed Symmetry-Breaking Constraints

Recall that a (partial) symmetry of an MCS defines equivalence classes on its (partial) equilibria through orbits. Symmetry breaking amounts to selecting some representatives from every equivalence class and formulating conditions, composed into a (distributed) symmetry-breaking constraint (SBC), that is only satisfied on those representatives. A *full* SBC selects exactly one representative from each orbit, otherwise we call an SBC *partial*. The most common approach is to order all elements from the solution space lexicographically, and to select the lexicographically smallest element, the *lex-leader*, from each orbit as its representative (see, for instance, [1,6]). A *lex-leader symmetry-breaking constraint* (LL-SBC) is an SBC that is satisfied only on the lex-leaders of orbits. Given an MCS $M = (C_1, \dots, C_n)$ with logics L_i over alphabet $\mathcal{A}_i$, we will assume a total ordering $<_{\mathcal{A}}$ on the atoms $a_1, a_2, \dots, a_m$ in $\mathcal{A}$ and consider the induced lexicographic ordering on the (partial) belief states. Following [6], we obtain an LL-SBC by encoding a (distributed) *permutation constraint* (PC) for every permutation π, where

$$\text{PC}(\pi) = \textstyle\bigwedge_{1 \le i \le m} \left[\bigwedge_{1 \le j \le i-1} (a_j = a_j^\pi)\right] \to (a_i \le a_i^\pi).$$

By *chaining*, which uses atoms $c_{\pi,i}$, $1 < i \le m+1$ (which informally express that for some $i \le j \le m$ the implication fails if it did not for some $j < i$), we achieve a representation that is linear in the number of atoms [1]:

$$\begin{aligned} \text{PC}(\pi) &= (a_1 \le a_1^\pi) \land \neg c_{\pi,2}, \\ \neg c_{\pi,i} &\leftrightarrow ((a_{i-1} \ge a_{i-1}^\pi) \to (a_i \le a_i^\pi) \land c_{\pi,i+1}) \qquad 1 < i \le m, \\ \neg c_{\pi,m+1} &\leftrightarrow \top. \end{aligned}$$

In order to distribute the PC formula in M, given the total ordering $<_{\mathcal{A}}$, we define (the truth of) atoms $c_{\pi,i}$ in the contexts C_k such that $a_{i-1} \in \mathcal{A}_k$. Observe that, for each subformula, the atoms a_i, a_i^π and $c_{\pi,i+1}$ might be defined in a different context j, and their truth value has to be imported via bridge rules. We thus introduce auxiliary atoms a_i', $a_i'^\pi$, and $c_{\pi,i+1}'$ in C_k that resemble the truth of a_i, a_i^π, and $c_{\pi,i+1}$, respectively. Then we distribute $\text{PC}(\pi)$ to each context C_k for each $1 \le k \le n$ as follows:

$$\begin{aligned} \text{PC}(\pi) &= (a_1 \le a_1^\pi) \land \neg c_{\pi,2} && \text{if } a_1 \in \mathcal{A}_k, \\ \neg c_{\pi,i} &\leftrightarrow ((a_{i-1} \ge a_{i-1}^\pi) \to (a_i \le a_i^\pi) \land \neg c_{\pi,i+1}) && \text{if } a_{i-1}, a_i \in \mathcal{A}_k, \\ \neg c_{\pi,i} &\leftrightarrow ((a_{i-1} \ge a_{i-1}^\pi) \to (a_i' \le a_i'^\pi) \land \neg c_{\pi,i+1}') && \text{if } a_{i-1} \in \mathcal{A}_k, a_i \in \mathcal{A}_j, j \ne k, \\ \neg c_{\pi,m+1} &\leftrightarrow \top && \text{if } a_m \in \mathcal{A}_k, \\[1ex] a_i' &\leftarrow (j : a_i) && \text{if } a_{i-1} \in \mathcal{A}_k, a_i \in \mathcal{A}_j, j \ne k, \\ a_i'^\pi &\leftarrow (j : a_i^\pi) && \text{if } a_{i-1} \in \mathcal{A}_k, a_i \in \mathcal{A}_j, j \ne k, \\ c_{\pi,i+1}' &\leftarrow (j : c_{\pi,i+1}) && \text{if } a_{i-1} \in \mathcal{A}_k, a_i \in \mathcal{A}_j, j \ne k. \end{aligned}$$

The distributed PC can be adjusted to other logics as well. Exploiting detected symmetries has been studied, e.g., in the context of SAT [1,6], planning [9], and constraint programming [15]. For an ASP context C_k, we can express the distributed PC as follows:

$$\left.\begin{array}{l}\leftarrow a_1, \sim a_1^\pi\\ \leftarrow c_{\pi,2}\end{array}\right\}\text{ if } a_1 \in \mathcal{A}_k; \qquad \left.\begin{array}{l}c_{\pi,i} \leftarrow a_{i-1}, a'_i, \sim a'^{\pi}_i\\ c_{\pi,i} \leftarrow \sim a^\pi_{i-1}, a'_i, \sim a'^{\pi}_i\\ c_{\pi,i} \leftarrow a_{i-1}, c'_{\pi,i+1}\\ c_{\pi,i} \leftarrow \sim a^\pi_{i-1}, c'_{\pi,i+1}\end{array}\right\}\begin{array}{l}\text{if } a_i \in \mathcal{A}_j,\\ a_{i-1} \in \mathcal{A}_k,\\ j \neq k;\end{array}$$

$$\left.\begin{array}{l}c_{\pi,i} \leftarrow a_{i-1}, a_i, \sim a_i^\pi\\ c_{\pi,i} \leftarrow \sim a^\pi_{i-1}, a_i, \sim a_i^\pi\\ c_{\pi,i} \leftarrow a_{i-1}, c_{\pi,i+1}\\ c_{\pi,i} \leftarrow \sim a^\pi_{i-1}, c_{\pi,i+1}\end{array}\right\}\begin{array}{l}\text{if } a_i \in \mathcal{A}_k,\\ a_{i-1} \in \mathcal{A}_k;\end{array} \qquad \left.\begin{array}{r}a'_i \leftarrow (j : a_i)\\ a'^{\pi}_i \leftarrow (j : a_i^\pi)\\ c'_{\pi,i+1} \leftarrow (j : c_{\pi,i+1})\end{array}\right\}\begin{array}{l}\text{if } a_i \in \mathcal{A}_j,\\ a_{i-1} \in \mathcal{A}_k,\\ j \neq k;\end{array}$$

Here, $c_{\pi,i}$ is defined from $\neg c_{\pi,i} \leftrightarrow (\alpha \rightarrow \beta \wedge \neg c_{\pi,i+1})$ via $c_{\pi,i} \leftrightarrow (\alpha \wedge \neg\beta \vee \alpha \wedge c_{\pi,i+1})$ exploiting Clark completion and splitting $\alpha = a_{i-1} \leq a^\pi_{i-1}$ into the (overlapping) cases where a_{i-1} is true and a^π_{i-1} is false. We collect the newly introduced formulas in $kb_{k,\pi}$ and bridge rules in $br_{k,\pi}$ for each $1 \leq k \leq n$. The following correctness result can be shown, generalizing a similar result for ASP programs in [8].

Theorem 4. *Let π be a (partial) symmetry of an MCS $M = (C_1, \ldots, C_n)$ with ASP contexts C_i. A (partial) equilibrium of M satisfies PC(π) iff it is a (partial) equilibrium of $M(\pi) = (C_1(\pi), \ldots, C_n(\pi))$, where $C_k(\pi)$ extends C_k by $kb_k(\pi) = kb_k \cup kb_{k,\pi}$ and $br_k(\pi) = br_k \cup br_{k,\pi}$.*

This result generalizes to MCS having contexts C_i with (possibly heterogeneous) logics L_i that permit to encode PC via additional formulas in the knowledge base kb_i.

Example 5 (cont'd). Reconsider M from Example 1. Given the ordering $a <_{\mathcal{A}} b <_{\mathcal{A}} d <_{\mathcal{A}} e$, the permutation constraint to break the partial symmetry $\pi = (a\ b)\ (d\ e)$ is:

$$\left.\begin{array}{ll}\leftarrow a, \sim b & c_{\pi,2} \leftarrow b, d', \sim e'\\ \leftarrow c_{\pi,2} & c_{\pi,2} \leftarrow \sim a, d', \sim e'\\ & c_{\pi,2} \leftarrow b, c'_{\pi,3}\\ & c_{\pi,2} \leftarrow \sim a, c'_{\pi,3}\end{array}\right\} kb_{1,\pi}, \qquad \left.\begin{array}{r}d' \leftarrow (2 : d)\\ e' \leftarrow (2 : e)\\ c'_{\pi,3} \leftarrow (2 : c_{\pi,3})\end{array}\right\} br_{1,\pi}, \text{ and}$$

$kb_{2,\pi} = br_{2,\pi} = \emptyset$. One can check that $(\{b\}, \{e\}, \varepsilon)$, $(\emptyset, \{d, e, f\}, \varepsilon)$, and $(\emptyset, \{d, e, g\}, \varepsilon)$ are partial equilibria of $M(\pi)$ w.r.t C_1, and $(\{a\}, \{d\}, \varepsilon)$ is not (cf. Example 1) since $(\{a\}, \{d\}, \varepsilon) <_{\mathcal{A}} (\{b\}, \{e\}, \varepsilon)$.

The LL-SBC that breaks every (partial) symmetry in an MCS, denoted LL-SBC(Π), can now be constructed by conjoining all of its permutation constraints [6]. We can add LL-SBC(Π) to M, say $M(\Pi) = (C_1(\Pi), \ldots, C_n(\Pi))$, where $C_k(\Pi)$ extends C_k by $kb_k(\Pi) = kb_k \cup \bigcup_{\pi \in \Pi} kb_k(\pi)$ and $br_k(\Pi) = br_k \cup \bigcup_{\pi \in \Pi} br_k(\pi)$.

Breaking all symmetries may not speed up search because there are often exponentially many of them. A better trade-off may be provided by breaking enough symmetries [6]. We explore partial SBCs, i.e., we do not require that SBCs are satisfied by lex-leading assignments only (but we still require that all lex-leaders satisfy SBCs). Irredundant generators are good candidates because they cannot be expressed in terms of each other, and implicitly represent all symmetries. Hence, breaking all symmetry in a generating set can eliminate all problem symmetries.

8 Experiments

We present some results on breaking local symmetries in terms of irredundant generators for distributed nonmonotonic MCS with ASP logics. Experiments consider the DMCS system [7] and its optimized version DMCSOPT [3]. Both systems are using the ASP solver CLASP [10] as their core reasoning engine. However, in contrast to DMCS, DMCSOPT exploits the topology of an MCS, that is the graph where contexts are nodes and import relations define edges, using decomposition techniques and minimises communication between contexts by projecting partial belief states to relevant atoms. We compare the average response time and the number of solutions under symmetry breaking, denoted as DMCS$^\pi$ and DMCSOPT$^\pi$, respectively, on benchmarks versus direct application of the respective systems. All tests were run on a 2×1.80 GHz PC under Linux, where each run was limited to 180 seconds. Our benchmarks stem from [3] and include random MCSs with various fixed topologies that should resemble the context dependencies of realistic scenarios. Experiments consider MCS instances with ordinary (D) and zig-zag (Z) diamond stack, house stack (H), and ring (R). A diamond stack combines multiple diamonds in a row, where ordinary diamonds (in contrast to zig-zag diamonds) have no connection between the 2 middle contexts. A house consists of 5 nodes with 6 edges such that the ridge context has directed edges to the 2 middle contexts, which form with the 2 base contexts a cycle with 4 edges. House stacks are subsequently built using the basement nodes as ridges for the next houses.

Table 1 shows some experimental results on calculating equilibria w.r.t. a randomly selected starting context of MSC with n contexts, where n varies between 9 and 151. Each context has an alphabet of 10 atoms, exports at most 5 atoms to other contexts, and has a maximum of 5 bridge rules with at most 2 bridge literals. First, we confirm the results of Bairakdar *et al.* [3], i.e., DMCSOPT can handle larger sizes of MCSs more efficiently than DMCS. Second, evaluating the MCS instances with symmetry breaking

Table 1. Completed runs (10 random instances each): avg. running time (secs) vs. timeouts

		DMCS		DMCS$^\pi$		DMCSOPT		DMCSOPT$^\pi$	
	n	time	#t.out	time	#t.out	time	#t.out	time	#t.out
D	10	1.90		0.46		0.54		0.35	
	13	62.12	4	32.21	2	1.38		0.98	
	25	—	10	—	10	16.12		11.72	
	31	—	10	—	10	84.02	1	58.95	
H	9	7.54		1.89		0.33		0.20	
	13	88.85	6	63.98	2	0.60		0.35	
	41	—	10	—	10	1.38		0.95	
	101	—	10	—	10	5.48		3.58	
R	10	0.36		0.26		0.15		0.12	
	13	22.41	1	5.11		0.19		0.16	
Z	10	6.80		3.24		0.62		0.37	
	13	57.58	3	42.93	3	1.03		0.68	
	70	—	10	—	10	18.87		9.98	
	151	—	10	—	10	51.10		30.15	

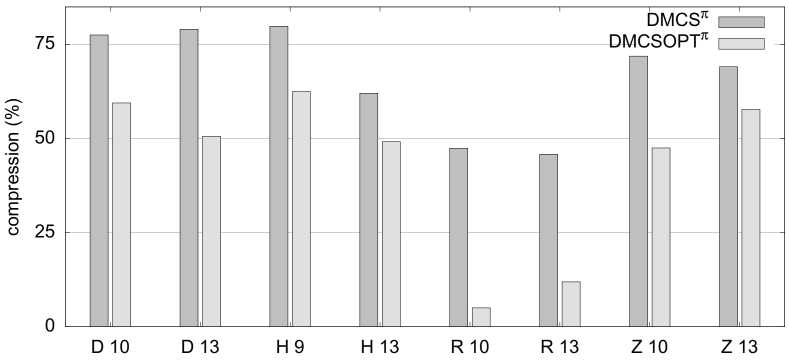


Fig. 3. Avg. compression of the solution space using local symmetry breaking w. irred. generators

compared to the direct application of either DMCS or DMCSOPT yields improvements in response time throughout all tested topologies. In fact, symmetry breaking always leads to better runtimes, and in some cases, returns solutions to problems which are otherwise intractable within the given time.

Fig. 3 presents the average compression of the solution space achieved by symmetry breaking. While the results for DMCS$^\pi$ range between 45% and 80%, the impact of symmetry breaking within DMCSOPT on the number of solutions varies between 5% and 65%. We explain the latter with the restriction of DMCSOPT to relevant atoms defined by the calling context.

9 Conclusion

We have presented a method for distributed symmetry detection and breaking for MCS. In particular, we have designed a distributed algorithm such that each context computes its own (partial) symmetries and communicates them with another for exchanging partial symmetries in order to compute symmetries of the system as a whole. Distributed symmetry-breaking constraints prevent an evaluation engine from ever visiting two points in the search space that are equivalent under the symmetry they represent. We have instantiated symmetry detection and symmetry breaking for MCS with ASP contexts, i.e., we have reduced partial symmetry of an ASP context to the automorphism of a coloured graph and encode symmetry breaking constraints as a distributed logic program. Experiments on recent MCS benchmarks and show promising results. Future work concerns a join operator for partial symmetries that preserves irredundant generators.

References

1. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Shatter: efficient symmetry-breaking for Boolean satisfiability. In: DAC 2003, pp. 836–839. ACM, New York (2003)
2. Babai, L.: Automorphism groups, isomorphism, reconstruction. In: Graham, R.L., Grötschel, M., Lovász, L. (eds.) Handbook of Combinatorics, vol. 2, pp. 1447–1540. Elsevier, Amsterdam (1995)

3. Bairakdar, S., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Decomposition of distributed nonmonotonic multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 24–37. Springer, Heidelberg (2010)
4. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
5. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI 2007, pp. 385–390. AAAI Press, Menlo Park (2007)
6. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: KR 1996, pp. 148–159. Morgan Kaufmann, San Francisco (1996)
7. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed nonmonotonic multi-context systems. In: KR 2010, pp. 60–70. AAAI Press, Menlo Park (2010)
8. Drescher, C., Tifrea, O., Walsh, T.: Symmetry-breaking answer set solving (2011) (to appear)
9. Fox, M., Long, D.: The detection and exploitation of symmetry in planning problems. In: IJCAI 1999, pp. 956–961. Morgan Kaufmann, San Francisco (1999)
10. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: *clasp*: A conflict-driven answer set solver. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 260–265. Springer, Heidelberg (2007)
11. Giunchiglia, F.: Contextual reasoning. Epistemologia, special issue on I Linguaggi e le Macchine 345, 345–364 (1992)
12. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics or: How we can do without modal logics. Artif. Intell. 65(1), 29–70 (1994)
13. Katsirelos, G., Narodytska, N., Walsh, T.: Breaking generator symmetry. In: SymCon 2009 (2009)
14. McCarthy, J.: Generality in artificial intelligence. Commun. ACM 30, 1030–1035 (1987)
15. Puget, J.-F.: Automatic detection of variable and value symmetries. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 475–489. Springer, Heidelberg (2005)

Splitting an Argumentation Framework

Ringo Baumann

Universität Leipzig, Johannisgasse 26, 04103 Leipzig, Germany
baumann@informatik.uni-leipzig.de

Abstract. Splitting results in non-mononotonic formalisms have a long tradition. On the one hand, these results can be used to improve existing computational procedures, and on the other hand they yield deeper theoretical insights into how a non-monotonic approach works. In the 90‘s Lifschitz and Turner [1,2] proved splitting results for logic programs and default theory. In this paper we establish similar results for Dung style argumentation frameworks (AFs) under the most important semantics, namely stable, preferred, complete and grounded semantics. Furthermore we show how to use these results in dynamical argumentation.

1 Introduction

Argumentation frameworks (AFs) as introduced in the seminal paper of Dung [3] are static. Since argumentation is a dynamic process, it is natural to investigate the dynamic behavior of AFs. In recent years the first publications appeared that deal with the problem of dynamical argumentation. One main direction in this field of research is to study the problem of how extensions of an AF may change if new (old) arguments and/or attack relations are added (deleted) (see, e.g. [4] and the references therein). A further question in this context is how to construct the extensions of an expanded AF by using the (already computed) acceptable sets of arguments of the initial AF. A solution to this problem is obviously of high interest from a computational point of view, especially in case of a huge number of arguments and attacks between them.

In 1994 Lifschitz and Turner [1] published splitting results for logic programs. They have shown that, under certain conditions, a logic program P can be split into two parts P_1 and P_2 such that the computation of an answer set can be considerably simplified: one computes an answer set E_1 of P_1, uses E_1 to modify P_2, computes an answer set E_2 of the modification of P_2 and then simply combines E_1 and E_2. We conveyed this idea to Dung‘s argumentation frameworks. It turns out that for stable semantics the result is similar to logic programs. However, for preferred, complete and grounded semantics, a more sophisticated modification is needed which takes into account that arguments may be neither accepted nor refuted in extensions.

Our definition of a splitting is closely connected with a special class of expansions, so-called *weak expansions*. This interrelation allows us to transfer our splitting results into the field of dynamical argumentation.

The paper is organized as follows: Section 2 reviews the necessary definitions at work in argumentation frameworks. The third section introduces new concepts like: Splitting, Expansion, Reduct, Undefined Set and Modification. Section 4, the main part of this paper, contains the splitting results for stable, preferred, complete and grounded semantics. Furthermore we compare the splitting theorem with a former monotonicity result. In section 5 we turn to dynamical argumentation. We concentrated on two issues: How to reuse already computed extensions and new terms of equivalence between two AFs. Finally, in section 6 we discuss related results and present our conclusions. Note that we will omit some proofs due to limited space. The full version of this paper is available at www.informatik.uni-leipzig.de/∼baumann/.

2 Preliminaries

We start with a brief review of the relevant definitions in argumentation theory.

Definition 1. *An argumentation framework $\mathcal{A}$ is a pair (A, R), where A is a non-empty finite set whose elements are called arguments and $R \subseteq A \times A$ a binary relation, called the attack relation.*

If $(a, b) \in R$ holds we say that a attacks b, or b is attacked by a. In the following we consider a fixed countable set $\mathcal{U}$ of arguments, called the *universe*. Quantified formulae refer to this universe and all denoted sets are finite subsets of $\mathcal{U}$ or $\mathcal{U} \times \mathcal{U}$ respectively. We introduce the union for two AFs $\mathcal{F} = (A_F, R_F)$ and $\mathcal{G} = (A_G, R_G)$ as expected, namely $\mathcal{F} \cup \mathcal{G} = (A_F \cup A_G, R_F \cup R_G)$. Furthermore we will use the following abbreviations.

Definition 2. *Let $\mathcal{A} = (A, R)$ be an AF, B and B' subsets of A and $a \in A$. Then*

1. $(B, B') \mathrel{\bar{\in}} R \Leftrightarrow_{def} \exists b \exists b' : b \in B \wedge b' \in B' \wedge (b, b') \in R$,
2. *a is defended by B in $\mathcal{A}$* $\Leftrightarrow_{def} \forall a' : a' \in A \wedge (a', a) \in R \rightarrow (B, \{a'\}) \mathrel{\bar{\in}} R$,
3. *B is conflict-free in $\mathcal{A}$* $\Leftrightarrow_{def} (B, B) \mathrel{\bar{\notin}} R$,
4. $cf(\mathcal{A}) = \{C \,|\, C \subseteq A, C$ *conflict-free in* $\mathcal{A}\}$.

Semantics of argumentation frameworks specify certain conditions for selecting subsets of a given AF $\mathcal{A}$. The selected subsets are called *extensions*. The set of all extensions of $\mathcal{A}$ under semantics $\mathcal{S}$ is denoted by $\mathcal{E}_{\mathcal{S}}(A)$. We consider the classical (stable, preferred, complete, grounded [3]) and the ideal semantics [5].

Definition 3. *Let $\mathcal{A} = (A, R)$ be an AF and $E \subseteq A$. E is a*

1. *stable extension ($E \in \mathcal{E}_{st}(\mathcal{A})$) iff*
   $E \in cf(\mathcal{A})$ *and for every* $a \in A \backslash E$, $(E, \{a\}) \mathrel{\bar{\in}} R$ *holds,*
2. *admissible extension*[1] *($E \in \mathcal{E}_{ad}(\mathcal{A})$) iff*
   $E \in cf(\mathcal{A})$ *and each* $a \in E$ *is defended by* E *in* $\mathcal{A}$,

[1] Note that it is more common to speak about admissible sets instead of the admissible semantics. For reasons of unified notation we used the uncommon version.

3. *preferred extension (i.e. $E \in \mathcal{E}_{pr}(\mathcal{A})$) iff* $E \in \mathcal{E}_{ad}(\mathcal{A})$ *and for each* $E' \in \mathcal{E}_{ad}(\mathcal{A})$, $E \not\subset E'$ *holds,*
4. *complete extension ($E \in \mathcal{E}_{co}(\mathcal{A})$) iff* $E \in \mathcal{E}_{ad}(\mathcal{A})$ *and for each* $a \in A$ *defended by* E *in* $\mathcal{A}$, $a \in E$ *holds,*
5. *grounded extension ($E \in \mathcal{E}_{gr}(\mathcal{A})$) iff* $E \in \mathcal{E}_{co}(\mathcal{A})$ *and for each* $E' \in \mathcal{E}_{co}(\mathcal{A})$, $E' \not\subset E$ *holds,*
6. *ideal extension of* $\mathcal{A}$ *($E \in \mathcal{E}_{id}(\mathcal{A})$) iff* $E \in \mathcal{E}_{ad}(\mathcal{A})$, $E \subseteq \bigcap_{P \in \mathcal{E}_{pr}(\mathcal{A})} P$ *and for each* $A \in \mathcal{E}_{ad}(\mathcal{A})$ *w.t.p.* $A \subseteq \bigcap_{P \in \mathcal{E}_{pr}(\mathcal{A})} P$ *holds* $E \not\subset A$.

3 Formal Foundation

In this section we will develop the technical tools which are needed to prove the splitting results.

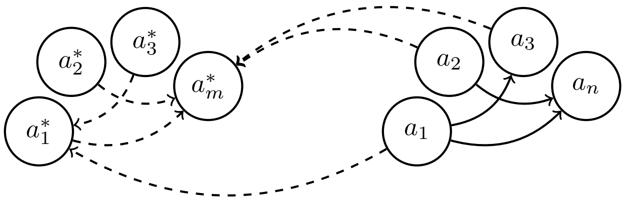
3.1 Splitting and Expansion

Definition 4. *Let* $\mathcal{A}_1 = (A_1, R_1)$ *and* $\mathcal{A}_2 = (A_2, R_2)$ *be AFs such that* $A_1 \cap A_2 = \emptyset$. *Let* $R_3 \subseteq A_1 \times A_2$. *We call the tuple* $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ *a splitting of the argumentation framework* $\mathcal{A} = (A_1 \cup A_2, R_1 \cup R_2 \cup R_3)$.

For short, a splitting of a given AF $\mathcal{A}$ is a partition in two disjoint AFs $\mathcal{A}_1$ and $\mathcal{A}_2$ such that the remaining attacks between $\mathcal{A}_1$ and $\mathcal{A}_2$ are restricted to a single direction. In [6] we studied the dynamical behavior of extensions of an AF. Therefore we introduced a special class of expansions of AFs, so-called *normal expansions*. *Weak* and *strong expansions* are two different subclasses of these expansions. After a short review of the definitions we will show that these kinds of expansions and the introduced splitting definition are in a sense two sides of the same coin. This observation allows us to convey splitting results into dynamical argumentation and vice versa.

Definition 5. *An AF* $\mathcal{A}^*$ *is an expansion of AF* $\mathcal{A} = (A, R)$ *iff* $\mathcal{A}^*$ *can be represented as* $(A \cup A^*, R \cup R^*)$ *for some nonempty* A^* *disjoint from* A *and some (possibly empty)* R^* *disjoint from* R . *Such an expansion is called to be*

1. *normal ($\mathcal{A} \prec^N \mathcal{A}^*$) iff* $\forall ab\ ((a,b) \in R^* \rightarrow a \in A^* \vee b \in A^*)$,
2. *strong ($\mathcal{A} \prec^N_S \mathcal{A}^*$) iff* $\mathcal{A} \prec^N \mathcal{A}^*$ *and* $\forall ab\ ((a,b) \in R^* \rightarrow \neg(a \in A \wedge b \in A^*))$,
3. *weak ($\mathcal{A} \prec^N_W \mathcal{A}^*$) iff* $\mathcal{A} \prec^N \mathcal{A}^*$ *and* $\forall ab\ ((a,b) \in R^* \rightarrow \neg(a \in A^* \wedge b \in A))$.



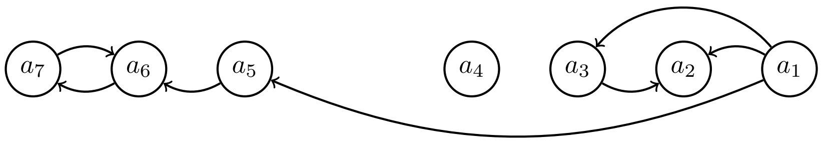
The figure above illustrates a weak expansion[2]. The dashed arrows represent the additional attack relation R^*. The following proposition establishes the connection between splittings and weak expansions. Note that this property is pretty obvious. Being aware of this fact, we still present it in the form of a proposition.

Proposition 1. *If $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ is a splitting of $\mathcal{A}$, then $\mathcal{A}$ is a weak expansion of $\mathcal{A}_1$. Vice versa, if $\mathcal{A} = (A, R)$ is a weak expansion of $\mathcal{A}_1 = (A_1, R_1)$, then $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ with $\mathcal{A}_2 = (A \backslash A_1, R \cap (A \backslash A_1 \times A \backslash A_1))$ and $R_3 = R \cap (A_1 \times A \backslash A_1)$ is a splitting of $\mathcal{A}$.*

3.2 Reduct, Undefined Set and Modification

Now we turn to the central definitions of our paper. The main goal is to establish a connection between the extensions of an AF $\mathcal{A}$ and a given splitting of it. Consider therefore the following example.

Example 1. Let $(\mathcal{A}_1, \mathcal{A}_2, \{(a_1, a_5)\})$ be a splitting of the AF $\mathcal{A}$ below, where $\mathcal{A}_1 = (\{a_1, a_2, a_3, a_4\}, \{(a_1, a_2), (a_1, a_3), (a_3, a_2)\})$ and
$\mathcal{A}_2 = (\{a_5, a_6, a_7\}, \{(a_5, a_6), (a_6, a_7), (a_7, a_6)\})$.



There are two stable extensions of $\mathcal{A}$, namely $E_1 = \{a_1, a_4, a_6\}$ and $E_2 = \{a_1, a_4, a_7\}$. Furthermore we observe that $E' = \{a_1, a_4\}$ and $E'' = \{a_5, a_7\}$ are the unique stable extensions of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Note that we cannot *reconstruct* the extensions E_1 and E_2 out of the extensions E' and E''. This is not very surprising because we do not take into account the attack (a_1, a_5). If we delete the argument a_5 in $\mathcal{A}_2$ which is attacked by E' and then compute the stable extensions in the reduced AF $\mathcal{A}_{2,red} = (\{a_6, a_7\}, \{(a_6, a_7), (a_7, a_6)\})$ we get the "missing" singletons $\{a_6\}$ and $\{a_7\}$. That means E_1 and E_2 are unions of the extensions of $\mathcal{A}_1$ and a reduced version of $\mathcal{A}_2$.

We will see that this observation holds in general for the stable semantics. The following definition of a reduct captures the intuitive idea.

Definition 6. *Let $\mathcal{A} = (A, R)$ be an AF, A' a set disjoint from A, $S \subseteq A'$ and $L \subseteq A' \times A$. The (S, L)-reduct of $\mathcal{A}$, denoted $\mathcal{A}^{S,L}$ is the AF*

$$\mathcal{A}^{S,L} = (A^{S,L}, R^{S,L})$$

where

$$A^{S,L} = \{a \in A \mid (S, \{a\}) \not\in L)\}$$

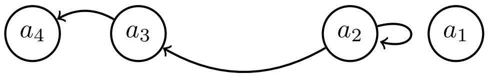
[2] The term is inspired by the fact that added arguments never attack previous arguments (*weak* arguments).

and

$$R^{S,L} = \{(a,b) \in R \mid a,b \in A^{S,L}\}.$$

The intuitionally described reduced version of the AF $\mathcal{A}_2$ in example 1 can be formalized exactly in the following way: $\mathcal{A}_{2,red} = (\{a_6, a_7\}, \{(a_6, a_7), (a_7, a_6)\}) = \mathcal{A}_2^{E',\{(a_1,a_5)\}}$. Unfortunately it turns out that the reduct used above does not obtain the desired properties for other semantics we are interested in. Here is a counterexample:

Example 2. Consider the AF $\mathcal{A} = (\{a_1, a_2, a_3, a_4\}, \{(a_2, a_2), (a_2, a_3), (a_3, a_4)\})$. $\mathcal{A}$ has a splitting $(\mathcal{A}_1, \mathcal{A}_2, \{(a_2, a_3)\})$ with $\mathcal{A}_1 = (\{a_1, a_2\}, \{(a_2, a_2)\})$ and $\mathcal{A}_2 = (\{a_3, a_4\}, \{(a_3, a_4)\})$.



$E_1 = \{a_1\}$ is the unique preferred, complete and grounded extension of $\mathcal{A}$. The same holds for the AF $\mathcal{A}_1$, i.e. $E' = \{a_1\}$. Consider now the $(E', \{(a_2, a_3)\})$-reduct of $\mathcal{A}_2$, that is $\mathcal{A}_2^{E',\{(a_2,a_3)\}} = \mathcal{A}_2$. The reduct establishes the unique extension $E'' = \{a_3\}$ for all considered semantics. Yet the union of E' and E'' differs from E_1.

The problem stems from the fact that the distinction between those arguments which are not in the extension *because they are refuted* (attacked by an accepted argument) and those not in the extension *without being refuted* is not taken care of. The former have no influence on $\mathcal{A}_2$. However, the latter - which we will call undefined in contrast to the refuted ones - indeed have an influence on $\mathcal{A}_2$, as illustrated in the example: the fact that a_2 is undefined in E' leads to the undefinedness of both a_3 and a_4, and this is not captured by the reduct. To overcome this problem, we introduce a simple modification. We can enforce undefinedness of a_3 (and thus of a_4) in $\mathcal{A}_2$ by introducing a self-attack for a_3. More generally, whenever there is an undefined argument a in the extension of the first AF which attacks an argument b in the second AF, we modify the latter so that b is both origin and goal of the attack.

Definition 7. *Let $\mathcal{A} = (A, R)$ be an AF, E an extension of $\mathcal{A}$. The set of arguments undefined with respect to E is*

$$U_E = \{a \in A \mid a \notin E, (E, \{a\}) \not\subseteq R\}.$$

Definition 8. *Let $\mathcal{A} = (A, R)$ be an AF, A' a set disjoint from A, $S \subseteq A'$ and $L \subseteq A' \times A$. The (S, L)-modification of $\mathcal{A}$, denoted $mod_{S,L}(\mathcal{A})$, is the AF*

$$mod_{S,L}(\mathcal{A}) = (A, R \cup \{(b,b) \mid \exists a : a \in S, (a,b) \in L\}).$$

Given a splitting $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ of $\mathcal{A}$, an extension E of $\mathcal{A}_1$ which leaves the set of arguments U_E undefined, we will use $mod_{U_E,R_3}(\mathcal{A}_2^{E,R_3})$ to compute what is missing from E. In case of example 2 we compute the extensions of the

$(\{a_2\},\{a_2,a_3\})$-modification of the $(E',\{(a_2,a_3)\})$-reduct of $\mathcal{A}_2$, i.e.

$$mod_{\{a_2\},\{a_2,a_3\}}\left(\mathcal{A}_2^{E',\{(a_2,a_3)\}}\right) = (\{a_3,a_4\},\{(a_3,a_3),(a_3,a_4)\}),$$

which establishes the empty set as the unique extension under all considered semantics. Hence the union of E' and $\emptyset$ equals the extension of the initial framework $\mathcal{A}$. Note that, although links are added, under all standard measures of the size of a graph (e.g. number of links plus number of vertices) we have $|\mathcal{A}_1| + |mod_{U_E,R_3}(\mathcal{A}_2^{E,R_3})| \leq |\mathcal{A}|$.

4 Splitting Results

Now we are going to present our formal results. Our splitting results show how to get extensions of the whole AF $\mathcal{A}$ with the help of a splitting. Furthermore we prove that our method is complete, i.e. all extensions are constructed this way. At first we will prove some simple properties of the introduced definitions.

Proposition 2. *Given an AF $\mathcal{A} = (A,R)$ which possesses a splitting $(\mathcal{A}_1,\mathcal{A}_2,R_3)$ s.t. $\mathcal{A}_1 = (A_1,R_1)$ and $\mathcal{A}_2 = (A_2,R_2)$, the following hold:*

1. $E_1 \in \mathcal{E}_{st}(\mathcal{A}_1) \Rightarrow mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3}) = \mathcal{A}_2^{E_1,R_3}$,
   (neutrality of the modification w.r.t. the stable reduct)
2. $E \in cf(\mathcal{A}) \Rightarrow E \cap A_1 \in cf(\mathcal{A}_1) \wedge E \cap A_2 \in cf(\mathcal{A}_2^{E\cap A_1,R_3})$,
   (preserving conflict-freeness[intersection])
3. $E_1 \in cf(\mathcal{A}_1) \wedge E_2 \in cf(mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3})) \Rightarrow E_1 \cup E_2 \in cf(\mathcal{A})$.
   (preserving conflict-freeness[union])

4.1 Monotonicity Result

In [6] we have proven the following monotonicity result. This theorem will be used to simplify parts of the proof of the splitting theorem. Furthermore we will see that the splitting theorem is a generalization of it.

Theorem 1. *Given an AF $\mathcal{A} = (A,R)$ and a semantics $\mathcal{S}$ satisfying directionality[3], then for all weak expansions $\mathcal{A}^*$ of $\mathcal{A}$ the following holds:*

1. $|\mathcal{E}_{\mathcal{S}}(\mathcal{A})| \leq |\mathcal{E}_{\mathcal{S}}(\mathcal{A}^*)|$,
2. $\forall E \in \mathcal{E}_{\mathcal{S}}(\mathcal{A})\ \exists E^* \in \mathcal{E}_{\mathcal{S}}(\mathcal{A}^*) : E \subseteq E^*$ *and*
3. $\forall E^* \in \mathcal{E}_{\mathcal{S}}(\mathcal{A}^*)\ \exists E_i \in \mathcal{E}_{\mathcal{S}}(\mathcal{A})\ \exists A_i^* \subseteq A^* : E^* = E_i \cup A_i^*$

Adding new arguments and their associated interactions may change the outcome of an AF in a nonmonotonic way. Accepted arguments may become unaccepted and vice versa. The theorem above specifies sufficient conditions (weak expansions + directionality principle) for monotonic behaviour w.r.t. justification state of an argument and cardinality of extensions. Remember that the

[3] Intuitively, the directionality principle prescribes that the acceptability of an argument a is determined only by its attackers (compare [7]).

admissible, complete, grounded, ideal and preferred semantics satisfy the directionality principle (compare [8]).

4.2 Splitting Theorem

Given a splitting of an AF $\mathcal{A}$, the general idea is to compute an extension E_1 of $\mathcal{A}_1$, reduce and modify $\mathcal{A}_2$ depending on what extension we got, and then compute an extension E_2 of the modification of the reduct of $\mathcal{A}_2$. The resulting union of E_1 and E_2 is an extension of $\mathcal{A}$. The second part of theorem proves the completeness of this method, i.e. all extensions are constructed this way.

Theorem 2. *($\sigma \in \{st, ad, pr, co, gr\}$) Let $\mathcal{A} = (A, R)$ be an AF which possesses a splitting $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ with $\mathcal{A}_1 = (A_1, R_1)$ and $\mathcal{A}_2 = (A_2, R_2)$.*

1. *If E_1 is an extension of $\mathcal{A}_1$ and E_2 is an extension of the (U_{E_1}, R_3)-modification of $\mathcal{A}_2^{E_1,R_3}$, then $E = E_1 \cup E_2$ is an extension of $\mathcal{A}$.*

$$\Big(E_1 \in \mathcal{E}_\sigma(\mathcal{A}_1) \wedge E_2 \in \mathcal{E}_\sigma(mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3})) \Rightarrow E_1 \cup E_2 \in \mathcal{E}_\sigma(\mathcal{A})\Big)$$

2. *If E is an extension of $\mathcal{A}$, then $E_1 = E \cap A_1$ is an extension of $\mathcal{A}_1$ and $E_2 = E \cap A_2$ is an extension of the (U_{E_1}, R_3)-modification of $\mathcal{A}_2^{E_1,R_3}$.*

$$\Big(E \in \mathcal{E}_\sigma(\mathcal{A}) \Rightarrow E \cap A_1 \in \mathcal{E}_\sigma(\mathcal{A}_1) \wedge E \cap A_2 \in \mathcal{E}_\sigma(mod_{U_{E\cap A_1},R_3}(\mathcal{A}_2^{E\cap A_1,R_3}))\Big)$$

Proof. **(stable)**(1.) by prop. 2.3 we got the conflict-freeness of $E_1 \cup E_2$ in $\mathcal{A}$; we now show that $E_1 \cup E_2$ attacks all outer arguments, i.e. for every $a \in (A_1 \cup A_2)\backslash(E_1 \cup E_2)$ holds: $(E_1 \cup E_2, \{a\}) \mathrel{\bar{\in}} R_1 \cup R_2 \cup R_3$; let a be an element of $A_1\backslash(E_1 \cup E_2)$, thus a is attacked by E_1 because $E_1 \in \mathcal{E}_{st}(\mathcal{A}_1)$ holds; let a be an element of $A_2\backslash(E_1 \cup E_2)$; we have to consider two cases because A_2 is the disjoint union of $\big\{a \in A_2 \,\big|(E_1, \{a\}) \mathrel{\bar{\notin}} R_3 \big\} \cup \{a \in A_2 \,|(E_1, \{a\}) \mathrel{\bar{\in}} R_3 \}$; if a is an element of the second set we have nothing to show; let a be an element the first set, namely $\big\{a \in A_2 \,\big|(E_1, \{a\}) \mathrel{\bar{\notin}} R_3 \big\} = A_2^{E_1,R_3}$; thus a is attacked by E_2 because $E_2 \in \mathcal{E}_{st}(mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3}))$ and $mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3}) = \mathcal{A}_2^{E_1,R_3}$ (prop. 2.1) holds;
(2.) at first we will show that $E \cap A_1 = E_1$ is a stable extension of $\mathcal{A}_1$, i.e. $E_1 \in \mathcal{E}_{st}(\mathcal{A}_1)$; conflict-freeness w.r.t. $\mathcal{A}_1$ follows from prop. 2.2; we only have to show that for every $a \in A_1\backslash E_1$ holds: $(E_1, \{a\}) \mathrel{\bar{\in}} R_1$; assume not, i.e. $\exists a \in A_1\backslash(E \cap A_1) : (E \cap A_1, \{a\}) \mathrel{\bar{\notin}} R_1$; consequently $(E, \{a\}) \mathrel{\bar{\notin}} R_1 \cup R_2 \cup R_3$ and this contradicts $E \in \mathcal{E}_{st}(\mathcal{A})$;
with prop. 2.1 and 2.2 we get $E \cap A_2 = E_2 \in cf(mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3}))$; we now show that $E \cap A_2$ attacks all outer arguments, i.e. for every $a \in A_2^{E_1,R_3}\backslash E_2$ holds: $(E_2, \{a\}) \mathrel{\bar{\in}} R_2^{E_1,R_3}$; assuming the contrary, i.e. $\exists a \in A_2^{E_1,R_3}\backslash E_2$: $(E_2, \{a\}) \mathrel{\bar{\notin}} R_2^{E_1,R_3}$ leads directly to $(E, \{a\}) \mathrel{\bar{\notin}} R_1 \cup R_2 \cup R_3$ which contradicts the fact that E is a stable extension of $\mathcal{A}$; $(E, \{a\}) \mathrel{\bar{\notin}} R_3$ because $a \in \big\{a^* \in A_2 \,\big|\, (E_1, \{a^*\}) \mathrel{\bar{\notin}} R_3 \big\}$ holds; furthermore $(E, \{a\}) \mathrel{\bar{\notin}} R_2$ because

$R_2^{E_1,R_3} \subseteq R_2$ holds and the remaining attacks in R_2 do not contain attacks from $A_2^{E_1,R_3}$ to $A_2^{E_1,R_3}$; the R_1 - case is obvious because $a \in A_2$ holds
(admissible) (1.) admissible sets are conflict-free per definition, hence conflict-freeness of $E_1 \cup E_2$ in $\mathcal{A}$ is given by prop. 2.3;
we have to show that each element of $E_1 \cup E_2$ is defended by $E_1 \cup E_2$ in $\mathcal{A}$, i.e. for each $a \in E_1 \cup E_2$ holds: if $(b,a) \in R_1 \cup R_2 \cup R_3$, then $(E_1 \cup E_2, \{b\}) \bar{\in} R_1 \cup R_2 \cup R_3$; let a be an element of E_1; if a is attacked by an element b, then $b \in A_1$ and $(b,a) \in R_1$ holds, hence the admissibility of E_1 in $\mathcal{A}_1$ guarentees the defence of a by $E_1 \cup E_2$ in $\mathcal{A}$; let a be an element of E_2; we have to consider two cases, namely $b \in A_1$ and $b \in A_2$; assuming $b \in A_1$ yields $(b,a) \in R_3$; we have already shown the conflict-freeness of $E_1 \cup E_2$, hence b has to be an element of $A_1 \backslash E_1$; again two cases arise, either $b \in U_{E_1}$ or $b \notin U_{E_1}$; the first case is not possible because elements which are attacked by undefined arguments w.r.t. E_1 get additional self-attacks in the modification, hence these elements can not be in the conflict-free extension E_2; the second case, namely $b \notin U_{E_1}$ (and $b \notin E_1$) can be true but if a is attacked by b, than $(E_1, \{b\}) \bar{\in} R_1$ holds per definition of the undefined arguments w.r.t. E_1; consider now $b \in A_2$ and $(b,a) \in R_2$; we have to distinguish two cases, namely $b \in A_2^{E_1,R_3}$ or $b \notin A_2^{E_1,R_3}$; the counterattack of b by $E_1 \cup E_2$ in the first case is assured because E_2 defends its elements in $mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3})$, hence E_2 defends its elements in $\mathcal{A}_2^{E_1,R_3}$ (the deleted self-attacks do not change the defense-state of elements in E_2); that means there is only one case left, namely $(b,a) \in R_2$ and $b \notin A_2^{E_1,R_3}$, i.e. $b \in \{a \in A_2 \,|(E_1, \{a\}) \bar{\in} R_3\}$; hence b is counterattacked by E_1 which completes the proof that a is defended by $E_1 \cup E_2$ in $\mathcal{A}$;
(2.) using that admissible semantics satisfying directionality we conclude immediately $E \cap A_1 = E_1$ is an admissible extension of $\mathcal{A}_1$, i.e. $E_1 \in \mathcal{E}_{ad}(\mathcal{A}_1)$ (compare theorem 1.3);
now we want to show that $E \cap A_2 = E_2 \in \mathcal{E}_{ad}(mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3}))$ holds; at first we note that E_2 is indeed a subset of $A_2^{E_1,R_3}$ (compare prop. 2.2); we now show the conflict-freeness of E_2 w.r.t. $mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3})$, i.e. $(E_2, E_2) \bar{\notin} R_2^{E_1,R_3} \cup \{(b,b) \mid a \in U_{E_1}, (a,b) \in R_3\}$; again prop. 2.2 justifies $(E_2, E_2) \bar{\notin} R_2^{E_1,R_3}$; $(E_2, E_2) \bar{\notin} \{(b,b) \mid a \in U_{E_1}, (a,b) \in R_3\}$ holds because if there is a b in E_2 which get a self-attack by the modification, than b has to be attacked by an undefined element $a \in U_{E_1}$; but this means that E does not defend its elements in $\mathcal{A}$ because a is per definition unattacked by E_1; at last we want to show that E_2 defends all its elements in $mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3})$; assume $a \in E_2 \wedge b \in A_2^{E_1,R_3} = \{b \in A_2 \mid (E_1, \{b\}) \bar{\notin} R_3\} \wedge (b,a) \in R_2^{E_1,R_3} \cup \{(b,b) \mid a \in U_{E_1}, (a,b) \in R_3\}$; we observe that $a \neq b$ holds, because i) E is conflict-free and $R_2^{E_1,R_3} \subseteq R_2$ holds and ii) the additional self-attacks of the modification do not involve elements of E_2 because assuming this contradicts again the fact that E defends all its elements in $\mathcal{A}$; thus $(b,a) \in R_2^{E_1,R_3} \subseteq R_2$ holds, consequently there is a $c \in E : (c,b) \in R_1 \cup R_2 \cup R_3$; it holds that $c \notin E \cap A_1$ because of $b \in A_2^{E_1,R_3}$, hence $c \in E \cap A_2 \wedge (c,b) \in R_2$ holds; this implies $(c,b) \in R_2^{E_1,R_3}$ which completes the proof

(preferred) (1.) we have to show that $E_1 \cup E_2 \in \mathcal{E}_{pr}(\mathcal{A})$, i.e. $E_1 \cup E_2$ is admissible (already shown since each preferred extension is admissible) and maximal w.r.t. the set inclusion; assume not, hence there is a $E^* \in \mathcal{E}_{ad}(\mathcal{A}) : E_1 \cup E_2 \subset E^*$; thus at least one of the following two cases is true: $E_1 \subset E^* \cap A_1$ or $E_2 \subset E^* \cap A_2$; assuming the first one contradicts the maximality of E_1 because $E^* \cap A_1$ is an admissible extension of $\mathcal{A}_1$; we observe that $E^* \cap A_1 = E_1$ holds; consider now $E_2 \subset E^* \cap A_2$; using the second part of the splitting theorem for admissible sets yields $E^* \cap A_2 \in \mathcal{E}_{ad}(mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3}))$ which contradicts the maximality of E_2; hence, we have proven that $E_1 \cup E_2 \in \mathcal{E}_{pr}(\mathcal{A})$ holds;
(2.) let E be an preferred extension of $\mathcal{A}$; using that the preferred semantics satisfies directionality we conclude $E \cap A_1 \in \mathcal{E}_{pr}(\mathcal{A}_1)$ (theorem 1.3); admissibility of $E \cap A_2$ w.r.t. $mod_{U_{E\cap A_1},R_3}(\mathcal{A}_2^{E\cap A_1,R_3})$ is obvious since every preferred extension is admissible (theorem 2.2 [admissible case]); assume now the existence of an $E_2^* \in \mathcal{E}_{ad}(mod_{U_{E\cap A_1},R_3}(\mathcal{A}_2^{E\cap A_1,R_3})) : E \cap A_2 \subset E_2^*$; thus $(E \cap A_1) \cup E_2^*$ is admissible w.r.t. $\mathcal{A}$ (theorem 2.1 [admissible case]) which contradicts the maximality of E and we are done
(complete) (1.) we have to show that $E_1 \cup E_2 \in \mathcal{E}_{co}(\mathcal{A})$, i.e. $E_1 \cup E_2$ is admissible (already shown since every complete extension is admissible) and for each $a \in A_1 \cup A_2$ which is defended by $E_1 \cup E_2$ in $\mathcal{A}$ holds: $a \in E_1 \cup E_2$; assume not, hence $\exists a \in (A_1 \cup A_2)\backslash(E_1 \cup E_2) : a$ is defended by $E_1 \cup E_2$ in $\mathcal{A}$; assuming that $a \in A_1\backslash(E_1 \cup E_2)$ holds contradicts $E_1 \in \mathcal{E}_{co}(\mathcal{A}_1)$; so let $a \in A_2\backslash(E_1 \cup E_2)$ be true; at first we observe that $a \in A_2^{E_1,R_3}$ holds because of the conflict-freeness of E_1 w.r.t. $\mathcal{A}_1$; we have to consider two attack-scenarios: a) a is attacked by arguments in $A_2\backslash A_2^{E_1,R_3}$ (and obviously defended by E_1 in $\mathcal{A}$); the reduct-relation do not contain such attacks, hence every "attack" is counterattacked by E_2 in $mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3})$; b) a is attacked by arguments in $A_2^{E_1,R_3}\backslash E_2$; hence it must be defended by elements of E_2 in $\mathcal{A}_2^{E_1,R_3}$, thus defended by E_2 in $mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3})$ because the corresponding attack-relation do not delete such counterattacks; altogether we have shown that $a \in E_2$ holds, hence $E_1 \cup E_2 \in \mathcal{E}_{co}(\mathcal{A})$ is proven;
(2.) assume $E \in \mathcal{E}_{co}(\mathcal{A})$; using that the complete semantics satisfies directionality we conclude $E \cap A_1 \in \mathcal{E}_{co}(\mathcal{A}_1)$ (theorem 1.3); admissibility of $E \cap A_2$ w.r.t. $mod_{U_{E\cap A_1},R_3}(\mathcal{A}_2^{E\cap A_1,R_3}))$ holds since complete extensions are admissible (theorem 2.2 [admissible case]); supposing $\exists a \in A_2^{E\cap A_1,R_3}\backslash E \cap A_2$: a is defended by $E \cap A_2$ in $mod_{U_{E\cap A_1},R_3}(\mathcal{A}_2^{E\cap A_1,R_3})$ contradicts the completeness of E in $\mathcal{A}$ because possible attackers of a are elements of $A_2^{E\cap A_1,R_3}$ which are counterattacked by $E \cap A_2$; these counterattacks are not added by the modification, hence a is defended by $E \cap A_2$ in $\mathcal{A}_2^{E\cap A_1,R_3}$; furthermore a is defended by E in $\mathcal{A}$ (further attackers are counterattacked by $E \cap A_1$) and again we conclude $E \notin \mathcal{E}_{co}(\mathcal{A})$
(grounded) (1.) we have to show that $E_1 \cup E_2 \in \mathcal{E}_{gr}(\mathcal{A})$, i.e. $E_1 \cup E_2$ is a complete extension of $\mathcal{A}$ (already shown since each grounded extensions is complete) and furthermore it is minimal w.r.t. the set inclusion; assume not, hence there is a set $E^* \in \mathcal{E}_{co}(\mathcal{A})$: $E^* \subset E_1 \cup E_2$; we will show that the following two cases are impossible: i) $E^* \cap A_1 \subset E_1$ or ii) $E^* \cap A_2 \subset E_2$; the first case contradict

directly the minimality of E_1 w.r.t. $\mathcal{A}_1$; we observe that $E^* \cap A_1 = E_1$ holds, hence $E^* \cap A_2$ is a complete extension of $mod_{U_{E_1},R_3}(\mathcal{A}_2^{E_1,R_3})$ which contradicts the minimality of E_2;
(2.) let E be a grounded extension of $\mathcal{A}$; using that the grounded semantics satisfies the directionality principle we deduce directly $E \cap A_1 \in \mathcal{E}_{gr}(\mathcal{A}_1)$ (theorem 1.3); assume now the existence of $E_2^* \in \mathcal{E}_{co}(mod_{U_{E \cap A_1},R_3}(\mathcal{A}_2^{E \cap A_1,R_3})) : E_2^* \subset E \cap A_2$, thus $E_2^* \cup (E \cap A_1)$ is a complete extension of $\mathcal{A}$ and of course a proper subset of E (which contradicts the minimality of E) $\square$

The splitting theorem obviously strengthens the outcome of the monotonicity result for the admissible, preferred, grounded and complete semantics which all satisfy the directionality principle. We do not only know that an old belief set is contained in a new one and furthermore every new belief set is the union of an old one and a (possibly empty) set of new arguments but rather that every new belief set is the union of an old one and an extension of the corresponding modified reduct and vice versa. The cardinality inequality of the monotonicity result (theorem 1.1) can be strengthened in the following way.

Corollary 1. *Let* $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ *be a splitting of the argumentation framework* $\mathcal{A}^* = (A_1 \cup A_2, R_1 \cup R_2 \cup R_3)$ *and* $\sigma \in \{ad, pr, co, gr\}$. *The following inequality holds:*

$$|\mathcal{E}_\sigma(\mathcal{A})| \leq \sum_{E_i \in \mathcal{E}_\sigma(\mathcal{A})} \left| \mathcal{E}_\sigma \left(mod_{U_{E_i},R_3}(\mathcal{A}_2^{E_i,R_3}) \right) \right| = |\mathcal{E}_\sigma(\mathcal{A}^*)| \, .$$

5 Dynamical Argumentation

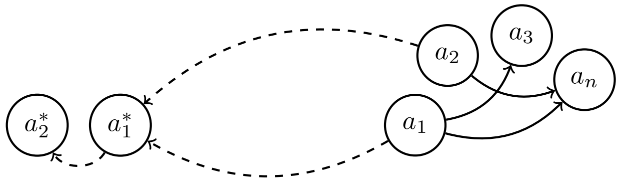
5.1 Computing Extensions

Since argumentation is a dynamic process, it is natural to investigate dynamic behavior in this context. Obviously the set of extensions of an AF may change if new arguments and their corresponding interactions are added. Computing the justification state of an argument from scratch each time new information is added is very inefficient. Note that in general, new arguments occur as a response, i.e., an attack, to a former argument. In this situation the former extensions are not reusable because in [6] we have shown a possibility result concerning the problem of enforcing of extensions which proves that every conflict-free subset of former arguments may belong to a new extension.

The splitting results allow us to reuse already computed extensions in case of weak expansions. Being aware of the remark above, we emphasize that weak expansions are not only a theoretical situation in argumentation theory. The initial arguments may be arguments which advance higher values[4] than the further arguments. The following dynamical argumentation scenario exemplifies how to use our splitting results.

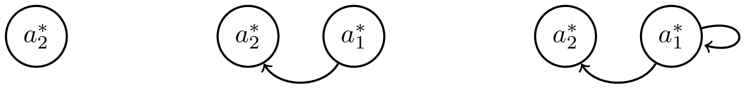
[4] Compare the idea of "attack-succeed" in Value Based Argumentation Frameworks [8].

Example 3. Given an AF $\mathcal{A} = (\{a_1, ..., a_n\}, R)$ and its set of extensions $\mathcal{E}_\sigma(\mathcal{A}) = \{E_1, ..., E_m\}$ $(\sigma \in \{pr, co, gr\})$. Consider now additional *new* arguments a_1^* and a_2^*, where a_1^* is attacked by the *old* arguments a_1 and a_2. Furthermore a_2^* is defeated by a_1^*.



What are the extensions of the expanded AF $\mathcal{A}^* = (A \cup \{a_1^*, a_2^*\}, R \cup \{(a_1, a_1^*), (a_2, a_1^*), (a_1^*, a_2^*)\})$? Since $\mathcal{A}^*$ is a weak expansion of $\mathcal{A}$ we may apply the splitting theorem. Given an extension E_i we construct the $(U_{E_i}, \{(a_1, a_1^*)(a_2, a_1^*)\})$-modification of $(\{a_1^*, a_2^*\}, \{(a_1^*, a_2^*)\})^{E_i, \{(a_1, a_1^*)(a_2, a_1^*)\}}$. The following three cases arise: (1) a_1 or a_2 is an element of E_i, (2) a_1 and a_2 are not in E_i and not in U_{E_i}, and (3) a_1 and a_2 are not in E_i and at least one of them is in U_{E_i}.

The AFs below are the resulting modifications in these three cases. In the first case the argument a_1^* disappears because a_1^* is attacked by an element of the extension E_i (reduct-definition). In the second and third case the arguments a_1^* and a_2^* survive because they are not attacked by E_i. Furthermore in the last case we have to add a selfloop for a_1^* since a_1 or a_2 are undefined (= not attacked) w.r.t. E_i.



The resulting preferred, complete and grounded extensions of the modifications are easily determinable, namely $\{a_2^*\}$ in the first case, $\{a_1^*\}$ in the second and the empty set in the last case. Now we can construct extensions of the expanded AF $\mathcal{A}^*$ by using the already computed extensions of $\mathcal{A}$, namely (1) $E_i \cup \{a_2^*\} \in \mathcal{E}_\sigma(\mathcal{A}^*)$, (2) $E_i \cup \{a_1^*\} \in \mathcal{E}_\sigma(\mathcal{A}^*)$, and (3) $E_i \in \mathcal{E}_\sigma(\mathcal{A}^*)$. Due to the completeness of the splitting method we constructed all extensions of $\mathcal{A}^*$.

We want to remark that the splitting results also provide a new possibility to compute extensions in a static AF. This work is still in progress and is not part of this paper.

5.2 Terms of Equivalence

Oikarinen and Woltran [9] extended the notion of equivalence between two AFs (which holds, if they possess the same extensions) to strong equivalence. Strong equivalence between to AFs $\mathcal{F}$ and $\mathcal{G}$ is fullfilled if for all AFs $\mathcal{H}$ holds that $\mathcal{F}$ conjoined with $\mathcal{H}$ and $\mathcal{G}$ conjoined with $\mathcal{H}$ are equivalent. Furthermore they establish criteria to decide strong equivalence. These characterizations are based on syntactical equality of so-called *kernels*.

The following definition weakens the strong equivalence notion w.r.t. weak expanions. We will present a characterization for stable semantics[5].

Definition 9. *Two AFs $\mathcal{F}$ and $\mathcal{G}$ are weak expansion equivalent to each other w.r.t. a semantics σ, in symbols $\mathcal{F} \equiv^{\sigma}_{\prec^N_W} \mathcal{G}$, iff for each AF $\mathcal{H}$ s.t.*

- $\mathcal{F} = \mathcal{F} \cup \mathcal{H}$ *or* $\mathcal{F} \prec^N_W \mathcal{F} \cup \mathcal{H}$ *and*
- $\mathcal{G} = \mathcal{G} \cup \mathcal{H}$ *or* $\mathcal{G} \prec^N_W \mathcal{G} \cup \mathcal{H}$,

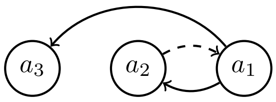
$\mathcal{E}_\sigma(\mathcal{F} \cup \mathcal{H}) = \mathcal{E}_\sigma(\mathcal{G} \cup \mathcal{H})$ *holds.*

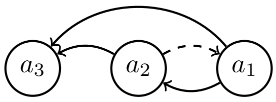
Proposition 3. *For any AFs $\mathcal{F} = (A_F, R_F)$ and $\mathcal{G} = (A_G, R_G)$: $\mathcal{F} \equiv^{st}_{\prec^N_W} \mathcal{G}$ iff*

- $A_F = A_G$ *and* $\mathcal{E}_{st}(\mathcal{F}) = \mathcal{E}_{st}(\mathcal{G})$ *or*
- $\mathcal{E}_{st}(\mathcal{F}) = \mathcal{E}_{st}(\mathcal{G}) = \emptyset$.

It obviously holds that strong equivalence between two AFs $\mathcal{F}$ and $\mathcal{G}$ implies their weak expansion equivalence. The following example demonstrates that the converse does not hold.

Example 4. Given $\mathcal{F} = (\{a_1, a_2, a_3\}, \{(a_1, a_2), (a_1, a_3)\})$,
$\mathcal{G} = (\{a_1, a_2, a_3\}, \{(a_1, a_2), (a_1, a_3), (a_2, a_3)\})$ and $\mathcal{H} = (\{a_1, a_2\}, \{(a_2, a_1)\})$.





Obviously we have $\mathcal{F} \equiv^{st}_{\prec^N_W} \mathcal{G}$ since $A_F = A_G$ and $\mathcal{E}_{st}(\mathcal{F}) = \mathcal{E}_{st}(\mathcal{G}) = \{a_1\}$ holds. On the other hand we have $\mathcal{E}_{st}(\mathcal{F} \cup \mathcal{H}) = \{\{a_1\}, \{a_2, a_3\}\}$ and $\mathcal{E}_{st}(\mathcal{G} \cup \mathcal{H}) = \{\{a_1\}, \{a_2\}\}$. Hence they are not strong equivalent.

6 Related Work and Conclusions

In this paper, we provided splitting results for Dung-style AFs under the most important semantics, namely stable, preferred, complete and grounded semantics. In a nutshell, the results show that each extension E of a splitted argumentation framework $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, R_3)$ is equal to the union of an extension E_1 of $\mathcal{A}_1$ and an extension E_2 of a modification (w.r.t. E_1 and R_3) of $\mathcal{A}_2$.

In [10] Baroni et al. introduced a general recursive schema for argumentation semantics. Furthermore they have shown that all admissibility-based semantics are covered by this definition. The great benefit of this approach is that the extensions of an AF $\mathcal{A}$ can be incrementally constructed by the extensions along its strongly connected components.

A directed graph is strongly connected if there is a path from each vertex to every other vertex. The SCCs of a graph are its maximal strongly connected

[5] The remaining semantics are left for future work.

subgraphs. Contracting every SCC to a single vertex leads to an acyclic graph. Hence every SCC-decomposition can be easily transformed into a splitting[6]. Conversely, a given splitting $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ simplifies the computation of SCCs because every SSC is either in $\mathcal{A}_1$ or $\mathcal{A}_2$. In this sense our results are certainly related to the SCC-approach. However, there are some important differences at various levels:

1. there is a subtle, yet relevant difference on the technical level: whereas the approach of Baroni et al. is based on a **generalized** theory of abstract argumentation (see subsection 5.1 in [10]), we stick to Dung's original approach and use an adequate modification in addition to the reduct to establish our results;
2. we provide theoretical insights about the relationship of the extensions of an **arbitrary** splitted AF; the parts into which an AF is split may be, but do not necessarily have to be SCCs;
3. whereas a major motivation in [10] was the identification of new semantics satisfying SCC-recursiveness, our primary intent is to carry over our results to **dynamical argumentation** like new terms of equivalence.

In section 4 we illustrated how to carry over our splitting results to dynamical argumentation. A number of papers appeared in this field of research. However, the possibility of reusing extensions has not received that much attention yet. A mentionable work in this context is [4]. Cayrol et al. proposed a typology of revisions (one new argument, one new interaction). Furthermore they proved sufficient conditions for being a certain revision type.

In future work we would like to study in detail the mentioned terms of equivalence between two AFs $\mathcal{A}$ and $\mathcal{B}$, i.e. what are sufficient and necessary conditions for their weak (strong, normal) expansion equivalence w.r.t. a semantics σ.

References

1. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: Principles of Knowledge Representation, pp. 23–37. MIT Press, Cambridge (1994)
2. Turner, H.: Splitting a default theory. In: Proc. AAAI 1996, pp. 645–651 (1996)
3. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and $n-$person games. Artificial Intelligence 77, 321–357 (1995)
4. Cayrol, C., Dupin de Saint-Cyr, F., Lagasquie-Schiex, M.-C.: Revision of an argumentation system. In: Proc. KR 2008, pp. 124–134 (2008)
5. Dung, P.M., Mancarella, P., Toni, F.: A dialectic procedure for sceptical, assumption-based argumentation. In: Proc. COMMA 2006, pp. 145–156. IOS Press, Liverpool (2006)
6. Baumann, R., Brewka, G.: Expanding Argumentation Frameworks: Enforcing and Monotonicity Results. In: Proc. COMMA 2010, pp. 75–86. IOS Press, Amsterdam (2010)

[6] Take the union of the initial nodes of the decomposition ($= \mathcal{A}_1$) and the union of the remaining subgraph ($= \mathcal{A}_2$).

7. Bench-Capon, T.: Value Based Argumentation Frameworks. In: Benferhat, S., Giunchiglia, E. (eds.) Proc. NMR 2002, Toulouse, France, pp. 443–445 (2002)
8. Baroni, P., Giacomin, M.: Evaluation and comparison criteria for extension-based argumentation semantics. In: Proc. COMMA 2006, pp. 157–168. IOS Press, Amsterdam (2006)
9. Oikarinen, E., Woltran, S.: Characterizing Strong Equivalence for Argumentation Frameworks. In: Proc. KR 2010, pp. 123–133. AAAI Press, Menlo Park (2010)
10. Baroni, P., Giacomin, M., Guida, G.: SCC-recursiveness: a general schema for argumentation semantics. Artificial Intelligence 168(1-2), 162–210 (2005)

Reactive Answer Set Programming

Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub⋆

Institut für Informatik, Universität Potsdam

Abstract. We introduce the first approach to Reactive Answer Set Programming, aiming at reasoning about real-time dynamic systems running online in changing environments. We start by laying the theoretical foundations by appeal to module theory. With this, we elaborate upon the composition of the various offline and online programs in order to pave the way for stream-driven grounding and solving. Finally, we describe the implementation of a reactive ASP solver, *oclingo*.

1 Introduction

Answer Set Programming (ASP; [1]) has become a popular declarative problem solving paradigm, facing a growing number of increasingly complex applications. So far, however, ASP systems are designed for offline usage, lacking any online capacities. We address this shortcoming and propose a reactive approach to ASP that allows us to implement real-time dynamic systems running online in changing environments. This new technology paves the way for applying ASP in many new challenging areas, dealing with agents, (ro)bots, policies, sensors, etc. The common ground of these areas is reasoning about dynamic systems incorporating online data streams.

For capturing dynamic systems, we take advantage of *incremental logic programs* [2], consisting of a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter t ranging over the natural numbers. In view of this, we sometimes denote P and Q by $P[t]$ and $Q[t]$. The base program B is meant to describe static knowledge, independent of parameter t. The role of P is to capture knowledge accumulating with increasing t, whereas Q is specific for each value of t. Roughly speaking, we are interested in finding an answer set of the program $B \cup \bigcup_{1\leq j\leq i} P[t/j] \cup Q[t/i]$ for some (minimum) integer $i \geq 1$.

As a motivating example, consider a very simple elevator controller accepting requests to go to a certain floor whenever it is not already at this floor. At each step, the elevator moves either up or down by one floor. If it reaches a floor for which a request exists, it serves the request automatically until its goal to serve all requests is fulfilled. This functionality is specified by the incremental logic program $(B, P[t], Q[t])$ in Fig. 1[1]. The answer set of the program $B \cup P[t/1] \cup Q[t/1]$ is $B \cup \{atFloor(2, 1), goal(1)\}$[2]. The elevator moves one floor and sees its goal fulfilled because there were no requests.

Observe that atoms of the form $request(F, t)$, representing incoming requests, are not defined by $P[t]$; that is, they do not occur in the head of any rule in $P[t]$. In fact,

⋆ Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

[1] We use shorthands '$1\{\dots\}1$', '..', ';', and ':' following the syntax of *gringo* (cf. [3]).

[2] For simplicity, we identify facts with atoms.

$$B = \left\{ \begin{array}{r} \mathit{floor}(1..3) \leftarrow \\ \mathit{atFloor}(1,0) \leftarrow \end{array} \right\}$$

$$P[t] = \left\{ \begin{array}{r} 1\,\{\mathit{atFloor}(F{-}1;F{+}1,t)\}\,1 \leftarrow \mathit{atFloor}(F,t{-}1), \mathit{floor}(F) \\ \leftarrow \mathit{atFloor}(F,t), \mathit{not}\ \mathit{floor}(F) \\ \mathit{requested}(F,t) \leftarrow \mathit{request}(F,t), \mathit{floor}(F), \mathit{not}\ \mathit{atFloor}(F,t) \\ \mathit{requested}(F,t) \leftarrow \mathit{requested}(F,t{-}1), \mathit{floor}(F), \mathit{not}\ \mathit{atFloor}(F,t) \\ \mathit{goal}(t) \leftarrow \mathit{not}\ \mathit{requested}(F,t) : \mathit{floor}(F) \end{array} \right\}$$

$$Q[t] = \{ \leftarrow \mathit{not}\ \mathit{goal}(t) \}\ .$$

Fig. 1. Incremental logic program for elevator control

requests are coming from outside the system, and their occurrences cannot be foreseen within an incremental program. Assume we get the request

$$E[1] = \{\mathit{request}(3,1) \leftarrow\} \tag{1}$$

telling our controller that a request to serve floor 3 occurred at time 1. While adding $E[1]$ to the above program yields no answer set, we get one from program $B \cup P[t/1] \cup P[t/2] \cup Q[t/2] \cup E[1]$, in which the elevator takes two steps to move to the third floor[2]:

$$B \cup E[1] \cup \{\mathit{requested}(3,1), \mathit{atFloor}(2,1), \mathit{atFloor}(3,2), \mathit{goal}(2)\}\ . \tag{2}$$

In fact, reasoning is driven by successively arriving events. No matter when a request arrives, its logical time step is aligned with the ones used in the incremental program. In this way, an event like (1) complements the domain description in Fig. 1 and initiates the subsequent search for an answer set as in (2). The next answer set computation is started by the following event, that is, upon the next request. As a particular feature of this methodology, observe that some rules in an encoding like $P[t]$ in Fig. 1 stay inactive until they get triggered by an event as in (1) adding $\mathit{request}(3,1)$ as a fact.

Grounding and solving in view of possible yet unknown future events constitutes a major technical challenge. For guaranteeing redundancy-freeness, the continuous integration of new program parts has to be accomplished without reprocessing previously treated programs. Also, simplifications related to events must be suspended until they become decided. Once this is settled, our approach leaves room for various application scenarios. While the above example is inspired by Cognitive Robotics [5], our approach may just as well serve as a platform for Autonomous Agent Architectures [6], Policy reasoning [7], or Sensor Fusion in Ambient Artificial Intelligence [8]. All in all, our approach thus serves as a domain-independent framework providing a sort of middle-ware for specific application areas rather than proposing a domain-specific solution.

2 Background

This section provides a brief introduction of answer sets of logic programs with choice rules and integrity constraints (see [1,9] for details). A *rule* is an expression of the form

$$h \leftarrow a_1, \dots, a_m, \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ a_n \tag{3}$$

where a_i, for $1 \leq m \leq n$, is an *atom* of the form $p(t_1, \dots, t_k)$, and $t_1, \dots, t_k$ are terms, viz., constants, variables, or functions. For a rule r as in (3), the *head* h of r is either an atom, a *cardinality constraint* of the form $l\,\{h_1, \dots, h_k\}\,u$ in which l, u are integers and $h_1, \dots, h_k$ are atoms, or the special symbol $\bot$. If h is a cardinality constraint, we call r a *choice rule*, and an *integrity constraint* if $h = \bot$. We denote the atoms occurring in h by $head(r)$, i.e., $head(r) = \{h\}$ if h is an atom, $head(r) = \{h_1, \dots, h_k\}$ if $h = l\,\{h_1, \dots, h_k\}\,u$, and $head(r) = \emptyset$ if $h = \bot$. In the following, we sometimes write h_r to refer to the head h of r, and we skip $\bot$ when writing an integrity constraint. The atoms occurring positively and negatively, respectively, in the *body* of r are denoted by $body(r)^+ = \{a_1, \dots, a_m\}$ and $body(r)^- = \{a_{m+1}, \dots, a_n\}$.

A *logic program* R is a set of rules of the form (3). By $atom(R)$, we denote the set of all atoms occurring in R, and $head(R) = \bigcup_{r \in R} head(r)$ is the collection of head atoms in R. The set of all ground terms constructible from the constants and function symbols that occur in R (if there is no constant, just add an arbitrary one) forms the *Herbrand universe* of R. The *ground instance* of R, denoted by $grd(R)$, is the set of all ground rules constructible from rules $r \in R$ by substituting every variable in r with some element of the Herbrand universe of R[3]. A set X of ground atoms satisfies a ground rule r of the form (3) if $\{a_1, \dots, a_m\} \subseteq X$ and $\{a_{m+1}, \dots, a_n\} \cap X = \emptyset$ imply that $h \in X$ or $h = l\,\{h_1, \dots, h_k\}\,u$ and $l \leq |\{h_1, \dots, h_k\} \cap X| \leq u$. We call X a *model* of R if X satisfies every rule $r \in grd(R)$. The *reduct* of R relative to X is $R^X = \{a \leftarrow body(r)^+ \mid r \in grd(R), a \in head(r) \cap X, body(r)^- \cap X = \emptyset\}$; X is an *answer set* of R if X is a model of R such that no proper subset of X is a model of R^X.

3 Reactive Answer Set Programming

In this section, we augment the concept of an incremental logic program with asynchronous information, refining the statically available knowledge. To this end, we characterize the constituents of the combined logic program including schematic as well as online parts, below called online progression.

An *online progression* represents a stream of events and inquiries. While entire event streams are made available for reasoning, inquiries act as punctual queries.

Definition 1. *We define an online progression* $(E_i[e_i], F_i[f_i])_{i \geq 1}$ *as a sequence of pairs of logic programs* E_i, F_i *with associated positive integers* e_i, f_i.

An online progression is asynchronous in distinguishing stream positions like i from (logical) time stamps. Hence, each event E_i and inquiry F_i includes a particular time stamp e_i or f_i, respectively, indicated by writing $E_i[e_i]$ and $F_i[f_i]$. Such time stamps are essential for synchronization with parameters in the underlying (incremental) logic programs. Note that different events and/or inquiries may refer to the same time stamp.

Definition 2. *Let* $(E_i[e_i], F_i[f_i])_{1 \leq i \leq j}$ *be a finite online progression and* $(B, P[t], Q[t])$ *be an incremental logic program. We define*

[3] We also assume that built-ins of grounders like *lparse* and *gringo* (cf. [4,3]), such as arithmetic functions, are evaluated upon instantiation.

1. *the k-expanded logic program of $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ wrt $(B, P[t], Q[t])$ as*

$$R_{j,k} = B \cup \bigcup_{1\leq i\leq k} P[t/i] \cup Q[t/k] \cup \bigcup_{1\leq i\leq j} E_i[e_i] \cup F_j[f_j] \tag{4}$$

   for each k such that $1 \leq e_1, \ldots, e_j, f_j \leq k$, and
2. *a reactive answer set of $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ wrt $(B, P[t], Q[t])$ as an answer set of a k-expanded logic program $R_{j,k}$ of $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ for a (minimum) $k \geq 1$.*

The incremental program constitutes the offline counterpart of an online progression; it is meant to provide a general (schematic) description of an underlying dynamic system. The parameter k represents a valid horizon accommodating all occurring events and inquiries. Thus, it is bound from below by the time stamps occurring in the online progression. The goal is then to find a (minimum) horizon k such that $R_{j,k}$ has an answer set, often in view of satisfying the global query $Q[t/k]$. In addition, inquiries, specific to each j, can be used for guiding answer set search. Unlike this, the whole stream $(E_i[e_i])_{1\leq i\leq j}$ of events is taken into account. Observe that the number j of events is independent of the horizon k. Finally, it is important to note that the above definition of an expanded program is static because its parameters are fixed. The next section is dedicated to the online evolution of reactive logic programs, characterizing the transitions from $R_{j,k}$ to $R_{j+1,k}$ and $R_{j,k}$ to $R_{j,k+1}$ in terms of constituent programs.

4 Reactive Modularity

For providing a clear interface between the various programs and guaranteeing their compositionality, we build upon the concept of a *module* [10], $\mathbb{P}$, being a triple (P, I, O) consisting of a (ground) program P and sets I, O of ground atoms such that $I \cap O = \emptyset$, $atom(P) \subseteq I \cup O$, and $head(P) \subseteq O$. The elements of I and O are called *input* and *output* atoms, also denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$, respectively; similarly, we refer to P by $P(\mathbb{P})$. The *join* of two modules $\mathbb{P}$ and $\mathbb{Q}$, denoted by $\mathbb{P} \sqcup \mathbb{Q}$, is defined as the module

$$(\ P(\mathbb{P}) \cup P(\mathbb{Q})\ ,\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P}))\ ,\ O(\mathbb{P}) \cup O(\mathbb{Q})\ )\ ,$$

provided that $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and there is no strongly connected component in the positive dependency graph of $P(\mathbb{P}) \cup P(\mathbb{Q})$, i.e., $(atom(P(\mathbb{P}) \cup P(\mathbb{Q})), \{(a, b) \mid r \in P(\mathbb{P}) \cup P(\mathbb{Q}), a \in head(r), b \in body(r)^+\})$, that shares atoms with both $O(\mathbb{P})$ and $O(\mathbb{Q})$. A set X of atoms is an *answer set* of a module $\mathbb{P} = (P, I, O)$ if X is a (standard) answer set of $P \cup \{a \leftarrow \mid a \in I \cap X\}$; we denote the set of all answer sets of $\mathbb{P}$ by $AS(\mathbb{P})$. For two modules $\mathbb{P}$ and $\mathbb{Q}$, the *composition* of their answer sets is $AS(\mathbb{P}) \bowtie AS(\mathbb{Q}) = \{X_{\mathbb{P}} \cup X_{\mathbb{Q}} \mid X_{\mathbb{P}} \in AS(\mathbb{P}), X_{\mathbb{Q}} \in AS(\mathbb{Q}), X_{\mathbb{P}} \cap (I(\mathbb{Q}) \cup O(\mathbb{Q})) = X_{\mathbb{Q}} \cap (I(\mathbb{P}) \cup O(\mathbb{P}))\}$. The module theorem [10] shows that the semantics of $\mathbb{P}$ and $\mathbb{Q}$ is *compositional* if their join is defined, i.e., if $\mathbb{P} \sqcup \mathbb{Q}$ is well-defined, then $AS(\mathbb{P} \sqcup \mathbb{Q}) = AS(\mathbb{P}) \bowtie AS(\mathbb{Q})$.

For turning programs into modules, we follow [2] and associate in Definition 3 a (non-ground) program P and a set I of (ground) input atoms with a module, denoted by $\mathbb{P}(I)$, imposing certain restrictions on the ground program induced by P. To this end, for a ground program P and a set X of ground atoms, define $P|_X$ as

$$\{h_r \leftarrow body(r)^+ \cup L \mid r \in P, body(r)^+ \subseteq X, L = \{not\ c \mid c \in body(r)^- \cap X\}\}\ .$$

Note that $P|_X$ projects the bodies of rules in P to the atoms of X. If a body contains an atom outside X, either the corresponding rule or literal is removed, depending on whether the atom occurs positively or negatively. This allows us to associate (non-ground) programs with (ground) modules, as proposed in [2].

Definition 3. *Let P be a logic program and I be a set of ground atoms. We define $\mathbb{P}(I)$ as the module $(\,grd(P)|_Y, I, head(grd(P)|_X)\,)$, where $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)|_X)$.*

The full ground instantiation $grd(P)$ of P is projected onto inputs and atoms defined in $grd(P)$. The head atoms of this projection, viz., $head(grd(P)|_{I\cup head(grd(P))})$, serve as output atoms and are used to simplify $grd(P)$, sparing only input and output atoms.

Unlike offline incremental ASP [2], its online counterpart deals with external knowledge acquired asynchronously. When constructing a ground module, we can thus no longer expect all of its atoms to be defined by the (ground) rules inspected so far. Rather, atoms may be defined by an online progression later on. To accommodate this, potential additions need to be reflected and exempted from program simplifications, as usually applied wrt (yet) undefined atoms. To this end, we assume in the following each (non-ground) program P to come along with some set of explicit ground input atoms (cf. the `#external` declaration described in Section 5), referred to by I_P. Such atoms provide "hooks" for online progressions to later incorporate new knowledge into an existing program part. Note that we could simply let $I_P = \emptyset$ for all program slices P to resemble offline incremental ASP.

We make use of the join to formalize the compositionality of instantiated modules induced by the respective programs in Definition 2.

Definition 4. *We define an online progression $(E_i[e_i], F_i[f_i])_{i\geq 1}$ as modular wrt an incremental logic program $(B, P[t], Q[t])$, if the modules*

$$\begin{aligned}
\mathbb{P}_0 &= \mathbb{B}(I_B) & \mathbb{P}_n &= \mathbb{P}_{n-1} \sqcup \mathbb{P}[t/n](O(\mathbb{P}_{n-1}) \cup I_{P[t/n]}) \\
\mathbb{E}_0 &= (\emptyset, \emptyset, \emptyset) & \mathbb{E}_n &= \mathbb{E}_{n-1} \sqcup \mathbb{E}_n[e_n](O(\mathbb{P}_{e_n}) \cup O(\mathbb{E}_{n-1}) \cup I_{E_n[e_n]})
\end{aligned}$$
$$\mathbb{R}_{j,k} = \mathbb{P}_k \sqcup \mathbb{E}_j \sqcup \mathbb{Q}[t/k](O(\mathbb{P}_k) \cup I_{Q[t/k]}) \sqcup \mathbb{F}_j[f_j](O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I_{F_j[f_j]})$$

are well-defined for all $j, k \geq 1$ such that $e_1, \ldots, e_j, f_j \leq k$.

In detail, this definition inspects the joins $\mathbb{P}_n$ and $\mathbb{E}_n$ of instantiated cumulative modules obtained from $P[t/n]$ and $E_n[e_n]$, respectively, for all $n \geq 0$. The former takes the instantiation of the static program B as its base case, where the input I_B related to B is considered by the instantiation. A module $\mathbb{P}_{n-1}$ obtained in this way is then joined with the instantiation of $P[t/n]$ relative to the atoms defined by preceding cumulative program slices, viz., $O(\mathbb{P}_{n-1})$, and the specific inputs $I_{P[t/n]}$, thus obtaining the next combined module $\mathbb{P}_n$. Observe that this join is independent of an online progression, yet the inputs collected over successive join operations provide an interface for online progressions to refine the available knowledge.

The join of the instantiations of online progressions' cumulative parts $E_n[e_n]$ starts from the empty module $\mathbb{E}_0$, given that the first event is provided for $n = 1$. Then, $\mathbb{E}_{n-1}$ is joined with the instantiation of $E_n[e_n]$ relative to the defined atoms requested via e_n, viz., $O(\mathbb{P}_{e_n})$, the atoms defined by preceding cumulative parts of the online progression, i.e., $O(\mathbb{E}_{n-1})$, and finally the particular inputs $I_{E_n[e_n]}$. As before, the latter provide means to refine the information gathered in the combined module $\mathbb{E}_n$.

A full module $\mathbb{R}_{j,k}$ incorporates all information accumulated in $\mathbb{P}_k$ and $\mathbb{E}_j$ as well as the volatile (query) parts $Q[t/k]$ and $F_j[f_j]$ of the incremental logic program and the online progression, respectively. Their instantiations consider all atoms defined by cumulative incremental program slices up to k or f_j, i.e., $O(\mathbb{P}_k)$ or $O(\mathbb{P}_{f_j})$, respectively, the specific inputs $I_{Q[t/k]}$ and $I_{F_j[f_j]}$, and in the case of $F_j[f_j]$ also the atoms $O(\mathbb{E}_j)$ defined by events of the online progression. Note that $O(\mathbb{E}_j)$ is not used as input for instantiating $Q[t/k]$, which reflects its role of belonging to an incremental logic program that is independent of and also invariant under particular online progressions. However, the explicit inputs $I_{Q[t/k]}$ (and $I_{F_j[f_j]}$) still admit passing information between (volatile parts of) the incremental logic program and an online progression.

The condition characterizing modularity of incremental programs and online progressions is that $\mathbb{R}_{j,k}$ must be well-defined for all $j, k \geq 1$, viz., each instantiation must yield a module and each join must be defined, where the requirement $e_1, \dots, e_j, f_j \leq k$ makes sure that the slices of an incremental program requested in an online progression contribute to $\mathbb{R}_{j,k}$. However, note that k is not bound to be $\max\{e_1, \dots, e_j, f_j\}$; rather, it can be increased beyond that as needed (for obtaining an answer set).

As an example, let us instantiate the incremental logic program in Fig. 1. While its static and query part, B and $Q[t]$, respectively, do not make use of particular inputs ($I_B = I_{Q[t]} = \emptyset$), the incremental part $P[t]$ relies on atoms of the form $request(F, t)$, which are not defined by $P[t]$. Unlike offline incremental ASP, where undefined atoms do not belong to the instantiation of a program slice, they must now be preserved to react to asynchronous requests. Accordingly, we let $I_{P[t]} = \{request(1,t), request(2,t), request(3,t)\}$; here, the first argument of an input atom is a floor and the second is the incremental parameter. Given the described inputs, the following ground modules are derived from the incremental program in Fig. 1 and contribute to $\mathbb{R}_{1,2}$:

$$\mathbb{P}_0 = \mathbb{B}(\emptyset) \qquad\qquad = (\, B, \emptyset, head(B)\,)$$
$$\text{where } B = \{\, floor(1) \leftarrow \quad floor(2) \leftarrow \quad floor(3) \leftarrow \qquad atFloor(1,0) \leftarrow \}$$
$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{P}[t/1](O(\mathbb{P}_0) \cup I_{P[t/1]}) = \big(\, P(\mathbb{P}_0) \cup P_1, I_{P[t/1]}, O(\mathbb{P}_0) \cup head(P_1)\,\big)$$
$$\text{where } P_1 = \left\{ \begin{array}{rl} 1\ \{atFloor(0,1), atFloor(2,1)\}\ 1 & \leftarrow atFloor(1,0), floor(1) \\ & \leftarrow atFloor(0,1) \\ & \leftarrow atFloor(2,1), not\ floor(2) \\ requested(1,1) & \leftarrow request(1,1), floor(1) \\ requested(2,1) & \leftarrow request(2,1), floor(2), not\ atFloor(2,1) \\ requested(3,1) & \leftarrow request(3,1), floor(3) \\ goal(1) & \leftarrow not\ requested(1,1), \\ & \quad\ not\ requested(2,1), \\ & \quad\ not\ requested(3,1) \end{array} \right\}$$

Note that the program in $\mathbb{P}_1$, viz. $P_1 = grd(P[t/1])|_{head(grd(P[t/1]) \cup B) \cup I_{P[t/1]}}$, is obtained by simplifying $grd(P[t/1])$ relative to the output of the preceding module, thereby, sparing the inputs $I_{P[t/1]}$ from simplifications (cf. Definition 3 and 4). For instance, input atoms of the form $request(F, 1)$ are not eliminated from P_1, while ground rules including undefined non-input atoms of the form $requested(F, 0)$ in their positive bodies do not contribute to P_1. The same considerations apply to program P_2 of $\mathbb{P}_2$ below. That is, $P_2 = grd(P[t/2])|_{head(grd(P[t/2]))\cup P_1 \cup B)\cup I_{P[t/2]}}$ is obtained by simplifying $grd(P[t/2])$ relative to the preceding outputs, while sparing the inputs $I_{P[t/2]}$:

$$\begin{aligned}\mathbb{P}_2 &= \mathbb{P}_1 \sqcup \mathbb{P}[t/2](O(\mathbb{P}_1) \cup I_{P[t/2]})\\ &= \big(P(\mathbb{P}_1) \cup P_2, I(\mathbb{P}_1) \cup I_{P[t/2]}, O(\mathbb{P}_1) \cup head(P_2)\big)\end{aligned}$$

where
$$P_2 = \left\{\begin{array}{rl}
1\ \{atFloor(1,2), atFloor(3,2)\}\ 1 & \leftarrow atFloor(2,1), floor(2)\\
 & \leftarrow atFloor(1,2), not\ floor(1)\\
 & \leftarrow atFloor(3,2), not\ floor(3)\\
requested(1,2) & \leftarrow request(1,2), floor(1), not\ atFloor(1,2)\\
requested(2,2) & \leftarrow request(2,2), floor(2)\\
requested(3,2) & \leftarrow request(3,2), floor(3), not\ atFloor(3,2)\\
requested(1,2) & \leftarrow requested(1,1), floor(1), not\ atFloor(1,2)\\
requested(2,2) & \leftarrow requested(2,1), floor(2)\\
requested(3,2) & \leftarrow requested(3,1), floor(3), not\ atFloor(3,2)\\
goal(2) & \leftarrow not\ requested(1,2),\\
 & \ not\ requested(2,2),\\
 & \ not\ requested(3,2)
\end{array}\right\}$$

$$\mathbb{Q}[t/2](O(\mathbb{P}_2)) = (\{\leftarrow not\ goal(2)\}, O(\mathbb{P}_2), \emptyset)$$

To complete $\mathbb{R}_{1,2}$, we further join $\mathbb{P}_2$ and $\mathbb{Q}[t/2](O(\mathbb{P}_2))$ with the module $\mathbb{E}_1 = (\{request(3,1) \leftarrow\}, \emptyset, \{request(3,1)\})$ stemming from the online progression $(\{request(3,1) \leftarrow\}, \emptyset)$, capturing the request $E[1]$ in (1). In view of its five input atoms, $request(1,1)$, $request(2,1)$, $request(1,2)$, $request(2,2)$, and $request(3,2)$, the full module $\mathbb{R}_{1,2}$ has four answer sets, obtained by augmenting the answer set shown in (2) with an arbitrary subset of $\{request(2,1), request(3,2)\}$. (That is, fictitious requests along the way of the elevator do not preclude it from serving floor 3, as required in view of $request(3,1) \leftarrow$.) However, note that the answer set in (2) is the only one that does not assume any of the residual input atoms of $\mathbb{R}_{1,2}$ to hold.

Regarding the formal properties of (modular) incremental logic programs and online progressions, we have that the module theorem [10] applies to instantiated modules contributing to $\mathbb{R}_{j,k}$.

Proposition 1 (Compositionality). *Let $(B, P[t], Q[t])$ be an incremental logic program, $(E_i[e_i], F_i[f_i])_{i\geq 1}$ be an online progression, $j, k \geq 1$ be such that $e_1, \ldots, e_j, f_j \leq k$, and $\mathbb{R}_{j,k}$ as well as $\mathbb{P}_n, \mathbb{E}_n$, for $n \geq 0$, be as in Definition 4.*
If $(E_i[e_i], F_i[f_i])_{i\geq 1}$ is modular wrt $(B, P[t], Q[t])$, then we have that

$$\begin{aligned}AS(\mathbb{R}_{j,k}) \;=\; & AS(\mathbb{P}_0) \bowtie AS(\mathbb{P}[t/1](O(\mathbb{P}_0) \cup I_{P[t/1]})) \bowtie \cdots \bowtie\\
& AS(\mathbb{P}[t/k](O(\mathbb{P}_{k-1}) \cup I_{P[t/k]})) \bowtie\\
& AS(\mathbb{E}_0) \bowtie AS(\mathbb{E}_1[e_1](O(\mathbb{P}_{e_1}) \cup O(\mathbb{E}_0) \cup I_{E_1[e_1]})) \bowtie \cdots \bowtie\\
& AS(\mathbb{E}_j[e_j](O(\mathbb{P}_{e_j}) \cup O(\mathbb{E}_{j-1}) \cup I_{E_j[e_j]})) \bowtie\\
& AS(\mathbb{Q}[t/k](O(\mathbb{P}_k) \cup I_{Q[t/k]})) \bowtie\\
& AS(\mathbb{F}_j[f_j](O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I_{F_j[f_j]}))\,.\end{aligned}$$

Note that compositionality holds wrt instantiated modules obtained by passing information (output atoms) from one module to another as specified in Definition 4.

Another question of interest concerns conditions under which the answer sets of a module $\mathbb{R}_{j,k}$ match the ones of a k-expanded logic program $R_{j,k}$, being the union of incremental logic program slices and programs of an online progression (cf. Definition 2). The major difference between both constructions is that modules contributing to $\mathbb{R}_{j,k}$ are instantiated successively wrt an evolving Herbrand universe, while the (non-ground) programs of $R_{j,k}$ share a Herbrand universe. To this end, we next provide a sufficient condition under which incremental and single-pass grounding yield similar answer sets. The idea is to require that, in the successive construction of $\mathbb{R}_{j,k}$, atoms that can already be used before they become defined must be declared to be inputs.

We say that an incremental logic program $(B, P[t], Q[t])$ and an online progression $(E_i[e_i], F_i[f_i])_{i\geq 1}$ are *mutually revisable* if the following conditions hold for all $n \geq 1$:

1. $\mathit{atom}(\mathit{grd}(B)) \cap \mathit{head}(\mathit{grd}(\bigcup_{i\geq 1}(P[t/i] \cup Q[t/i] \cup E_i[e_i] \cup F_i[f_i]))) \subseteq I_B$,
2. $\mathit{atom}(\mathit{grd}(P[t/n])) \cap \mathit{head}(\mathit{grd}(\bigcup_{i>n} P[t/i] \cup \bigcup_{i\geq n} Q[t/i] \cup \bigcup_{i\geq 1}(E_i[e_i] \cup F_i[f_i]))) \subseteq I_{P[t/n]}$,
3. $\mathit{atom}(\mathit{grd}(Q[t/n])) \cap \mathit{head}(\mathit{grd}(\bigcup_{i\geq 1}(E_i[e_i] \cup F_i[f_i]))) \subseteq I_{Q[t/n]}$,
4. $\mathit{atom}(\mathit{grd}(E_n[e_n])) \cap \mathit{head}(\mathit{grd}(\bigcup_{i>e_n} P[t/i] \cup \bigcup_{i\geq e_n} Q[t/i] \cup \bigcup_{i>n} E_i[e_i] \cup \bigcup_{i\geq n} F_i[f_i])) \subseteq I_{E_n[e_n]}$, and
5. $\mathit{atom}(\mathit{grd}(F_n[f_n])) \cap \mathit{head}(\mathit{grd}(\bigcup_{i>f_n} P[t/i] \cup \bigcup_{i\geq f_n} Q[t/i])) \subseteq I_{F_n[f_n]}$.

Observe that atoms belonging to the ground instance of the static program B or a cumulative program slice $P[t/n]$ must be consumed as inputs, i.e., belong to I_B or $I_{P[t/n]}$, respectively, if they can be defined by subsequent cumulative or query programs, or by the online progression. The latter condition must likewise hold for a query program $Q[t/n]$, which can however ignore atoms defined by program slices $I_{P[t/i]}$, for $i > n$, because a different query program $Q[t/i]$ will then be used instead. For the programs $E_n[e_n]$ and $F_n[f_n]$ of the online progression, we similarly require in 4. and 5. that all atoms in their ground instances that can be defined by the incremental program in a step $i > e_n$ or $i > f_n$ (also $i = e_n$ or $i = f_n$ for $Q[t/i]$), respectively, must be contained in $I_{E_n[e_n]}$ or $I_{F_n[f_n]}$. The inputs of an event $E_n[e_n]$ also need to include atoms that can be defined later by the online progression, i.e., in $E_i[e_i]$ or $F_i[f_i]$ for $i > n$ (also $i = n$ for $F_i[f_i]$). In summary, if all requirements of mutual revisability are met, the instantiated modules in $\mathbb{R}_{j,k}$ are via their inputs susceptible to atoms defined subsequently, as in the case of instantiating the full collection $R_{j,k}$ of (non-ground) programs in a single pass.

The following result formalizes the correspondence between the answer sets of $R_{j,k}$ and the ones of $\mathbb{R}_{j,k}$ not including input atoms, provided that mutual revisability applies.

Proposition 2 (Instantiation). *Let $(B, P[t], Q[t])$ be an incremental logic program, $(E_i[e_i], F_i[f_i])_{i\geq 1}$ be a modular online progression wrt $(B, P[t], Q[t])$, $j, k \geq 1$ be such that $e_1, \ldots, e_j, f_j \leq k$, $R_{j,k}$ be the k-expanded logic program of $(E_i[e_i], F_i[f_i])_{1\leq i\leq j}$ wrt $(B, P[t], Q[t])$, and $\mathbb{R}_{j,k}$ be as in Definition 4.*

If $(B, P[t], Q[t])$ and $(E_i[e_i], F_i[f_i])_{i\geq 1}$ are mutually revisable, then we have that X is an answer set of $R_{j,k}$ iff X is an answer set of $\mathbb{R}_{j,k}$ such that $X \subseteq O(\mathbb{R}_{j,k})$.

Note that, by letting $I_{P[t]} = \{\mathit{request}(1, t), \mathit{request}(2, t), \mathit{request}(3, t)\}$, Proposition 2 applies to the incremental logic program in Fig. 1 along with the online progression

$(\{request(3,1) \leftarrow\}, \emptyset)$, capturing the request $E[1]$ in (1). In fact, atoms defined by $P[t/n]$ do not occur in B or $P[t/i]$ for any $1 \leq i < n$, so that $I_{P[t]}$ is sufficient to reflect facts representing asynchronously arriving requests in instantiated modules. Hence, the answer set in (2) is obtained both for $R_{1,2}$ and $\mathbb{R}_{1,2}$. In fact, it is the only answer set of $\mathbb{R}_{1,2}$ not including any of its residual input atoms, viz., $request(1,1)$, $request(2,1)$, $request(1,2)$, $request(2,2)$, and $request(3,2)$.

To see that incremental instantiation and single pass grounding yield, in general, different semantics, note that, if $I_{P[t]} = \emptyset$, the instantiated modules obtained from $P[t]$ in Fig. 1 do not include any rule containing an atom of the form $request(F,t)$ in the positive body. Then, the answer sets of $\mathbb{R}_{1,2}$ would not yield a schedule to satisfy a request given in an online progression, but merely provide possible moves of the elevator. Unlike this, a request like in (1) would still be served in an answer set of $R_{1,2}$.

5 The Reactive ASP Solver *oclingo*

We implemented a prototypical reactive ASP solver called *oclingo*, which is available at [11] and extends *iclingo* [2] with online functionalities. To this end, *oclingo* acts as a server listening on a port, configurable via its `--port` option upon start-up. Unlike *iclingo*, which terminates after computing an answer set of the incremental logic program it is run on, *oclingo* waits for client requests. To issue such requests, we implemented a separate controller program that sends online progressions to *oclingo* and displays answer sets received in return.

For illustrating the usage of *oclingo*, consider Table 1 displaying the source code representation (`elevator.lp`) of the incremental logic program in Fig. 1. Its three parts are distinguished via the declarations '`#base.`', '`#cumulative t.`', and '`#volatile t.`', respectively, where `t` serves as the parameter. Of particular interest is the declaration preceded by '`#external`', delineating the input to the cumulative part provided by future online progressions (cf. $I_{P[t/n]}$ in Definition 4). In fact, the

Table 1. `elevator.lp`

```
#base.
floor(1..3).
atFloor(1,0).

#cumulative t.
#external request(F,t) : floor(F).
1 { atFloor(F-1;F+1,t) } 1 :- atFloor(F,t-1), floor(F).
:- atFloor(F,t), not floor(F).
requested(F,t) :- request(F,t),      floor(F), not atFloor(F,t).
requested(F,t) :- requested(F,t-1), floor(F), not atFloor(F,t).
goal(t) :- not requested(F,t) : floor(F).

#volatile t.
:- not goal(t).
```

declaration instructs *oclingo* to not apply any simplifications in view of yet undefined instances of `request(F,t)`, where `F` is a floor.

After launching *oclingo* on file `elevator.lp`, it proceeds according to Algorithm 1, which is basically an extension of *iclingo*'s `isolve` algorithm [2]. The base part is grounded in Line 4 and added to the solver in Line 5. Then, the main loop starts by waiting for external knowledge (Line 8), passed to *oclingo* by a client. For instance, the external knowledge representing the online progression in (1) is provided as follows:

```
#step 1.     request(3,1).     #endstep.
```

Here '`#step 1.`' specifies the time stamp $m_1 = 1$, and E_1 is '`request(3,1).`', as signaled via '`#endstep.`' A program for F_1 could be provided by specifying rules after a '`#volatile.`' declaration, but this functionality is not yet supported by *oclingo*. If it were, note that m_1 is supposed to be the maximum of e_1 and f_1 (cf. Definition 1).

Algorithm 1. `osolve`

```
Input    : An incremental logic program $(B, P[t], Q[t])$.
Internal : A grounder GROUNDER and a solver SOLVER.
1  $i \leftarrow 0$
2  $i_{old} \leftarrow 0$
3  $j \leftarrow 0$
4  $P_0 \leftarrow$ GROUNDER.ground$(B)$
5  SOLVER.add$(P_0)$
6  loop
7      $j \leftarrow j + 1$
8      $(E_j, F_j, m_j) \leftarrow$ getExternalKnowledge()
9      while $i < m_j$ do
10         $i \leftarrow i + 1$
11         $P_i \leftarrow$ GROUNDER.ground$(P[t/i])$
12         SOLVER.add$(P_i)$
13     $O_j \leftarrow$ GROUNDER.ground$(E_j \cup F_j(\beta_j))$
14     SOLVER.add$(O_j \cup \{\leftarrow \beta_{j-1}\})$
15     repeat
16         if $i_{old} < i$ then
17             $Q_i \leftarrow$ GROUNDER.ground$(Q[t/i](\alpha_i))$
18             SOLVER.add$(Q_i \cup \{\leftarrow \alpha_{i_{old}}\})$
19             $i_{old} \leftarrow i$
20         $\mathcal{X} \leftarrow$ SOLVER.solve$(\{\alpha_i, \beta_j\})$
21         if $\mathcal{X} = \emptyset$ then
22             $i \leftarrow i + 1$
23             $P_i \leftarrow$ GROUNDER.ground$(P[t/i])$
24             SOLVER.add$(P_i)$
25     until $\mathcal{X} \neq \emptyset$
26     send$(\{X \setminus \{\alpha_i, \beta_j\} \mid X \in \mathcal{X}\})$
```

After receiving the external knowledge, since $i = 0 < m_1 = 1$ in Line 9, `osolve` proceeds by incrementing i and processing a first slice of the incremental program's cumulative part. This includes grounding $P[t/1]$ and adding the ground program to the solver. Similarly, in Line 13 and 14, E_1 (and $F_1(\beta_1) = \emptyset$) are grounded and added to the solver. The notation $F_j(\beta_j)$ indicates that a fresh atom β_j is inserted into the body of each rule in F_j, so that the inquiry can in step $j + 1$ be discarded in Line 14 via adding the integrity constraint $\leftarrow \beta_j$ to the solver (cf. [12,2]). Note that *oclingo* currently supports ground external input only, so that the "grounding" in Line 13 merely maps textual input to an internal representation.

The repeat loop starting in Line 15 is concerned with unrolling the incremental program in view of satisfying $Q[t]$. In our example, $Q[t/1](\alpha_1)$ is grounded and then added to the solver (Line 17 and 18), where a fresh atom α_i is used to mark volatile rules to enable their discarding via adding an integrity constraint $\leftarrow \alpha_{i_{old}}$ later on. (Note that the step number m_j passed as external knowledge may cause jumps of i in query programs $Q[t/i]$, which are not possible with $P[t]$ in view of the loop starting in Line 9, and i_{old} is used to address a volatile part becoming obsolete.) The solving accomplished in Line 20 checks for an answer set in the presence of Q_1, stipulating the absence of pending requests for the elevator at time step 1. Note that α_1 and β_1 are passed as assumptions (cf. [12,2]) to the solver, telling it that queries in Q_1 and F_1 must be fulfilled. On the other hand, *oclingo* makes sure that yet undefined input atoms, i.e., elevator requests that did not arrive, are not subject to "guessing." In this case, `request(1,1)` and `request(2,1)` must not belong to an answer set, as no such external knowledge has been provided.

Given the pending request for floor 3, no answer set is obtained in Line 20, i.e., $\mathcal{X} = \emptyset$. Thus, the next cumulative program slice, $P[t/2]$, is grounded and added to the solver (Line 23 and 24). Then, the repeat loop is re-entered, where the query $Q[t/2](\alpha_2)$ is added and $Q[t/1](\alpha_1)$ discarded (Line 17 and 18). Afterwards, the answer set in (2) is found in Line 20. As mentioned above, it contains `request(3,1)` as the only externally provided atom: although $Q[t/2](\alpha_2)$ would stay satisfied if `request(2,1)` and/or `request(3,2)` were assumed to be true, *oclingo* eliminates these options by disallowing undefined input atoms to hold. Finally, the obtained answer set (without α_2 and β_1) is sent back to the client for further processing (Line 26) before *oclingo* waits for new external knowledge in Line 8. In practice, this process terminates when the client sends '`#stop.`' (rather than '`#step` m_j`. ... #endstep.`') to *oclingo*.

As already described, the current version of *oclingo* does not yet support non-ground or volatile external input. Furthermore, it includes no modularity checks for successively obtained ground program slices (cf. Definition 3). As a consequence, it is the responsibility of the user to make sure that all programs are modularly composable (cf. Definition 4) in order to guarantee that the answer sets computed wrt the jth online program and the incremental program up to step k match the ones of the combined module $\mathbb{R}_{j,k}$ that do not assume residual inputs to hold. (The successive grounding performed by *oclingo* yields answer sets of $\mathbb{R}_{j,k}$ rather than of $R_{j,k}$ (cf. Definition 2); see Proposition 2 for sufficient conditions guaranteeing their correspondence.) Note that modularity between incremental and online programs is easiest achieved at the predicate level, primarily, by not using atoms over input predicates in the heads of rules in the incremental program; e.g., `elevator.lp` follows this methodology. Of course, one also ought to take the modularity of the incremental program, when it is unrolled, into account (cf. [2]).

Note that, in view of the incremental approach, the step counter i is never decreased within `osolve`. Hence, it does not admit a "step back in time" wrt successive online programs and can, in general, not guarantee the k in answer sets of $\mathbb{R}_{j,k}$ to be minimal. (The minimal k', k such that $\mathbb{R}_{j-1,k'}$ and $\mathbb{R}_{j,k}$ admit answer sets may be such that $k < k'$.) To support minimality, one could add optimization statements (like `#minimize` and `#maximize`) to incremental programs, which is a subject to future work.

The application-oriented features of *oclingo* also include declarations '`#forget t.`' in external knowledge to signal that yet undefined input atoms, declared at a step smaller or equal to `t` are no longer exempted from simplifications, so that they can be falsified irretrievably by the solver in order to compact its internal representation of accumulated incremental program slices (cf. [12,2]). Furthermore, *oclingo* supports an asynchronous reception of input, i.e., the call in Line 8 of Algorithm 1 is processed also if solving in Line 20, relative to a previous online program, is still ongoing. If new input arrives before solving is finished, the running solving process is aborted, and the solver is relaunched wrt the new external knowledge.

6 Further Case Studies

The purpose of our reactive framework is to provide a middle-ware for various application areas. For the sake of utility and versatility, we have conducted a set of assorted case studies, all of which are available at [11].

First of all, we have experimented with a more complex elevator control than given above, involving opening and closing doors as well as more elaborated control knowledge. The strategy is to never change directions, as long as there is an active request in a current direction. Also, this use case adds an external input indicating the respective position of the elevator. A second case study is an extension of the well-known blocksworld example [13]. Our extension allows for new, falling blocks thwarting previously computed states and/or plans. This scenario aims at studying dynamic adaptions to new (unexpected) situations. Our third use case deals with position tracking by means of sensor networks. In contrast to the above planning tasks, this scenario looks for histories and is thus directed backward in time. It is inspired by [14], where a person moves in a home environment (given as a 2D grid) involving doors, rooms, obstacles, walls, etc. Interestingly, missing sensor information may lead to alternative histories. Moreover, these histories may change with the arrival of further sensor readings.

Our last two scenarios deal with simple games. The first one considers the well-known Wumpus world [15]. An agent moves on a grid and tries to find gold in the dark, while avoiding pits and the Wumpus. The moving Wumpus is externally controlled, and the agent has to react to the bad Wumpus' smell. The latter is obtained through events within an online progression. This is a typical agent-oriented scenario in which an agent has to react in view of its changing environment. The second game-based use case implements a simplistic TicTacToe player. This scenario is interesting from a technical point of view because it allows for having two ASP players compete with each other. Interestingly, each move has to be communicated to both players in order to keep the game history coherent.

7 Discussion

We introduced the first genuinely reactive approach to ASP. For this purpose, we developed a module theory guaranteeing an incremental composition of programs, while avoiding redundancy in grounding and solving. Unlike offline incremental ASP [2], reactive ASP includes dedicated support of the input/output interface from (ground) module theory [10]; in practice, inputs can be declared conveniently at the predicate level. Our approach has a general, domain-independent nature and may thus serve as a middle-ware opening up numerous new reactive applications areas to ASP. To this end, we have implemented the reactive ASP solver *oclingo* and conducted a variety of case studies demonstrating the utility and versatility of our approach. The implementation along with all case studies are freely available at [11].

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 190–205. Springer, Heidelberg (2008)
3. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`, `http://potassco.sourceforge.net`
4. Syrjänen, T.: Lparse 1.0 user's manual, `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`
5. Reiter, R.: Knowledge in Action. The MIT Press, Cambridge (2001)
6. Balduccini, M., Gelfond, M.: The autonomous agent architecture. Newsletter ALP 23 (2010)
7. Son, T., Lobo, J.: Reasoning about policies using logic programs. In: ASP 2001. AAAI/The MIT Press (2001)
8. Mileo, A., Merico, D., Bisiani, R.: Non-monotonic reasoning supporting wireless sensor networks for intelligent monitoring: The SINDI system. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 585–590. Springer, Heidelberg (2009)
9. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
10. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: ECAI 2006, pp. 412–416. IOS Press, Amsterdam (2006)
11. `http://www.cs.uni-potsdam.de/wv/oclingo`
12. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in TCS 89(4) (2003)
13. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann, San Francisco (2004)
14. Mileo, A., Schaub, T., Merico, D., Bisiani, R.: Knowledge-based multi-criteria optimization to support indoor positioning. In: RCRA 2010, CEUR-WS.org (2010)
15. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson, London (2010)

Communicating ASP and the Polynomial Hierarchy

Kim Bauters[1,⋆], Steven Schockaert[1,⋆⋆], Dirk Vermeir[2], and Martine De Cock[1]

[1] Department of Applied Mathematics and Computer Science
Universiteit Gent, Krijgslaan 281, 9000 Gent, Belgium
{kim.bauters,steven.schockaert,martine.decock}@ugent.be
[2] Department of Computer Science
Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
dvermeir@vub.ac.be

Abstract. Communicating answer set programming is a framework to represent and reason about the combined knowledge of multiple agents using the idea of stable models. The semantics and expressiveness of this framework crucially depends on the nature of the communication mechanism that is adopted. The communication mechanism we introduce in this paper allows us to focus on a sequence of programs, where each program in the sequence may successively eliminate some of the remaining models. The underlying intuition is that of leaders and followers: each agent's decisions are limited by what its leaders have previously decided. We show that extending answer set programs in this way allows us to capture the entire polynomial hierarchy.

1 Introduction

Communicating answer set programming is an extension of answer set programming (ASP) in which a number of logic programs, with their own knowledge and reasoning capabilities, can communicate and cooperate with each other to solve the problem at hand. A number of different flavors of Communicating ASP have already been proposed in the literature (*e.g.* [1,8,21]). Communicating ASP is also closely related to the research on multi-context systems, where a group of agents can cooperate to find the solution of a global problem [4,20]. We start off with an introductory example.

Example 1. An employee ('E') needs a new printer ('P'). She has a few choices (loud or silent, stylish or dull), preferring silent and stylish. Her boss ('B') does not want an expensive printer, *i.e.* one that is both silent and stylish. We can

⋆ Funded by a joint Research Foundation-Flanders (FWO) project.
⋆⋆ Postdoctoral fellow of the Research Foundation-Flanders (FWO).

consider the communicating program $\mathcal{P} = \{E, B\}$ with:

$$P{:}\mathit{stylish} \leftarrow \mathit{not}\ P{:}\mathit{dull} \qquad P{:}\mathit{dull} \leftarrow \mathit{not}\ P{:}\mathit{stylish} \tag{1}$$

$$P{:}\mathit{silent} \leftarrow \mathit{not}\ P{:}\mathit{loud} \qquad P{:}\mathit{loud} \leftarrow \mathit{not}\ P{:}\mathit{silent} \tag{2}$$

$$E{:}\mathit{undesired} \leftarrow P{:}\mathit{dull} \qquad E{:}\mathit{undesired} \leftarrow P{:}\mathit{loud} \tag{3}$$

$$B{:}\mathit{expensive} \leftarrow P{:}\mathit{stylish}, P{:}\mathit{silent}. \tag{4}$$

Intuitively, the rule '$B{:}\mathit{expensive} \leftarrow P{:}\mathit{stylish}, P{:}\mathit{silent}$' expresses that the agent B believes that the printer is '*expensive*' when according to agent P it is '*stylish*' and '*silent*'. The rules in (1) and (2) encode the four possible printers, the rules in (3) and (4) encode the inclinations of the employee and boss, respectively. The answer sets of this program, *i.e.* those with global minimality, are [1]

$$M_1 = \{P{:}\mathit{sty}, P{:}\mathit{silent}, B{:}\mathit{exp}\} \qquad M_2 = \{P{:}\mathit{sty}, P{:}\mathit{loud}, E{:}\mathit{und}\}$$
$$M_3 = \{P{:}\mathit{dull}, P{:}\mathit{loud}, E{:}\mathit{und}\} \qquad M_4 = \{P{:}\mathit{dull}, P{:}\mathit{silent}, E{:}\mathit{und}\}$$

where for compactness we write $P{:}\mathit{sty}, E{:}\mathit{und}$ and $B{:}\mathit{exp}$ instead of $P{:}\mathit{stylish}$, $E{:}\mathit{undesired}$ and $B{:}\mathit{expensive}$, respectively. The answer sets with minimality for the agent B are M_2, M_3 and M_4, *i.e.* the answer sets that do not contain $B{:}\mathit{expensive}$. The only answer set with minimality for agent E is M_1, *i.e.* the one that does not contain $E{:}\mathit{undesired}$. Hence when we determine local minimality for communicating ASP, the order in which we determine the local minimality is important and induces a preference over the agents, *i.e.* it makes some agents more important than others. In this example, if the boss comes first, the employee no longer has the choice to pick M_1. This leaves her with the choice of either a dull or a loud printer, among which she has no preferences.

Answer set semantics is based on the idea of stable minimal models. When dealing with agents that can communicate, it becomes unclear how we should interpret the notion of minimality as will become clear later on. One option is to assume global minimality, *i.e.* we minimize over the conclusions of all the agents in the network. Another option is to assume minimality on the level of a single agent. Since it is not always possible to find a model that is minimal for all individual agents, the order in which we minimize over the agents matters, as the preceding example illustrates.

In this paper we introduce the notion of multi-focused answer sets of communicating ASP programs, which allow us to successively focus (*i.e.* minimize) on different agents. We study the resulting expressiveness and hence complete the picture sketched in [1] on the effect that adding communication has on the expressiveness of ASP. In [1], it was shown that the ability to communicate allows for simple programs (programs without negation-as-failure, which alone are only capable of expressing problems in P) to simulate normal programs (which do have negation-as-failure and are capable of expressing problems in NP). Furthermore, [1] introduced a new communication mechanism where one can "focus" on a single agent, allowing to express problems in Σ_2^{P}. In this paper, we go a step further by introducing multi-focused answer sets. Multi-focused answer sets allow us to focus successively on a number of agents instead of focusing on just one agent, and is therefore a generalization of the idea of focusing. As it turns out, using multi-focused answer set programs it is possible to express any problem in PSPACE. This means in particular that communicating ASP could be used to solve problems that are above the second level of the polynomial hierarchy, such as some forms of abductive reasoning [10] as well as PSPACE-complete problems such as STRIPS planning [6].

The remainder of this paper is organized as follows. In Section 2 we give the necessary background on answer set programming. In Section 3, we recall the syntax and semantics of communicating ASP. In Section 4 we introduce a generalization of focused answer sets [1], capable of expressing any Σ_n^{P} problem. We finish with Section 5 and Section 6 where we discuss related work and present our conclusion.

2 Background on Answer Set Programming

We first recall the basic concepts and results from ASP that are used in this paper. To define ASP programs, we start from a countable set of atoms and we define a *literal* l as an atom a or its classical negation $\neg a$. If L is a set of literals, we use $\neg L$ to denote the set $\{\neg l \mid l \in L\}$ where, by definition, $\neg\neg a = a$. A set of literals L is consistent if $L \cap \neg L = \emptyset$. An *extended literal* is either a literal or a literal preceded by *not* which we call the negation-as-failure operator. Intuitively we say that *not* l is true when we have no proof to support l. For a set of literals L, we use $not(L)$ to denote the set $\{not\ l \mid l \in L\}$.

A *normal rule* is an expression of the form $l \leftarrow (\alpha \cup not(\beta))$ with 'l' a literal called the head of the rule and $(\alpha \cup not(\beta))$ (interpreted as a conjunction) the body of the rule with α and β sets of literals. When the body is empty, the rule is called a *fact*. When the head is empty, the rule is called a *constraint*. In this paper, we do not consider constraints as they can readily be simulated[1]. A *normal program* P is a finite set of normal rules. The *Herbrand base* $\mathcal{B}_P$ of P is the set of atoms appearing in program P. A (partial) *interpretation* I of P is any consistent set of literals $L \subseteq (\mathcal{B}_P \cup \neg\mathcal{B}_P)$. I is total iff $I \cup \neg I = \mathcal{B}_P \cup \neg\mathcal{B}_P$. A *simple rule* is a normal rule without negation-as-failure in the body. A *simple program* P is a finite set of simple rules.

Answer sets are defined using the *immediate consequence operator* T_P for a simple program P *w.r.t.* an interpretation I as

$$T_P(I) = I \cup \{l \mid ((l \leftarrow \alpha) \in P) \wedge (\alpha \subseteq I)\}\,. \tag{5}$$

We use $P^\star$ to denote the fixpoint which is obtained by repeatedly applying T_P starting from the empty interpretation, *i.e.* the least fixpoint of T_P *w.r.t.* set inclusion. The interpretation $P^\star$ is the minimal model of P and is called the *answer set* of the simple program P.

The *reduct* P^I of a normal program P *w.r.t.* the interpretation I is defined as $P^I = \{l \leftarrow \alpha \mid (l \leftarrow \alpha \cup not(\beta)) \in P, \beta \cap I = \emptyset\}$. It is easy to see that the reduct P^I is a simple program. We say that I is an answer set of the normal program P iff $\left(P^I\right)^\star = I$, *i.e.* if I is the answer set of the reduct P^I.

[1] The constraint $(\leftarrow body)$ is simulated by $(fail \leftarrow not\ fail, body)$ with $fail$ a fresh atom.

3 Communicating Programs

Communication between ASP programs is based on a new kind of literal '$Q\!:\!l$' as in [1,3,16,20], where the underlying intuition is that of a function call or, in terms of agents, asking questions to other agents. If the literal l is not in the answer set of program Q then $Q\!:\!l$ is false; otherwise $Q\!:\!l$ is true. We present the semantics from [1] which are closely related to the minimal semantics in [3] and the semantics in [5].

Let $\mathcal{P}$ be a finite set of program names. A *$\mathcal{P}$-situated literal* is an expression of the form $Q\!:\!l$ with $Q \in \mathcal{P}$ and l a literal. A $\mathcal{P}$-situated literal $Q\!:\!l$ is called *Q-local*. For a set of $\mathcal{P}$-situated literals X and $Q \in \mathcal{P}$, we use $X_{\downarrow Q}$ to denote $\{l \mid Q\!:\!l \in X\}$, *i.e.* the projection of X on Q. An *extended $\mathcal{P}$-situated literal* is either a $\mathcal{P}$-situated literal or a $\mathcal{P}$-situated literal preceded by *not*. For a set of $\mathcal{P}$-situated literals X, we use $\mathit{not}(X)$ to denote the set $\{\mathit{not}\ Q\!:\!l \mid Q\!:\!l \in X\}$.

A *$\mathcal{P}$-situated normal rule* is an expression of the form $Q\!:\!l \leftarrow (\alpha \cup \mathit{not}(\beta))$ where $Q\!:\!l$ is called the head of the rule, and $\alpha \cup \mathit{not}(\beta)$ is called the body of the rule with α and β sets of $\mathcal{P}$-situated literals. A $\mathcal{P}$-situated normal rule $Q\!:\!l \leftarrow (\alpha \cup \mathit{not}\ (\beta))$ is called Q-local. A *$\mathcal{P}$-component normal program Q* is a finite set of Q-local $\mathcal{P}$-situated normal rules. Henceforth we shall use $\mathcal{P}$ both to denote the set of program names and to denote the set of actual $\mathcal{P}$-component normal programs. A *communicating normal program $\mathcal{P}$* is then a finite set of $\mathcal{P}$-component normal programs. A *$\mathcal{P}$-situated simple rule* is an expression of the form $Q\!:\!l \leftarrow \alpha$, *i.e.* a $\mathcal{P}$-situated normal rule without negation-as-failure in the body. A *$\mathcal{P}$-component simple program Q* is a finite set of Q-local $\mathcal{P}$-situated simple rules. A *communicating simple program $\mathcal{P}$* is a finite set of $\mathcal{P}$-component simple programs.

In the remainder of this paper we drop the $\mathcal{P}$-prefix whenever the set $\mathcal{P}$ is clear from the context. Whenever the name of the component normal program Q is clear, we write l instead of $Q\!:\!l$ for Q-local situated literals. Note that a communicating normal (resp. simple) program with only one component program thus trivially corresponds to a normal (resp. simple) program.

Similar as for a classical program, we can define the *Herbrand base* of a component program Q as the set of atoms occurring in Q-local situated literals in Q, which we denote as $\mathcal{B}_Q$. We then define $\mathcal{B}_{\mathcal{P}} = \left\{Q\!:\!a \mid Q \in \mathcal{P} \text{ and } a \in \bigcup_{R \in \mathcal{P}} \mathcal{B}_R\right\}$ as the Herbrand base of the communicating program $\mathcal{P}$.

Example 2. Consider the communicating simple program $\mathcal{P} = \{Q, R\}$ with the following situated rules:

$$Q\!:\!a \leftarrow R\!:\!a \qquad\qquad Q\!:\!b \leftarrow \qquad\qquad R\!:\!a \leftarrow Q\!:\!a.$$

$Q\!:\!a$, $Q\!:\!b$ and $R\!:\!a$ are situated literals. The situated simple rules $Q\!:\!a \leftarrow R\!:\!a$ and $Q\!:\!b \leftarrow$ are Q-local since we have $Q\!:\!a$ and $Q\!:\!b$ in the heads of these rules. The situated simple rule $R\!:\!a \leftarrow Q\!:\!a$ is R-local. Hence $Q = \{a \leftarrow R\!:\!a, b \leftarrow\}$ and $R = \{a \leftarrow Q\!:\!a\}$. Furthermore, we have that $\mathcal{B}_Q = \{a, b\}$, $\mathcal{B}_R = \{a\}$ and $\mathcal{B}_{\mathcal{P}} = \{Q\!:\!a, Q\!:\!b, R\!:\!a, R\!:\!b\}$.

We say that a (partial) interpretation I of a communicating program $\mathcal{P}$ is any consistent subset $I \subseteq (\mathcal{B}_\mathcal{P} \cup \neg\mathcal{B}_\mathcal{P})$. Given an interpretation I of a communicating normal program $\mathcal{P}$, the reduct Q^I for $Q \in \mathcal{P}$ is the component simple program obtained by deleting

- each rule with an extended situated literal '*not* $R{:}l$' such that $R{:}l \in I$;
- each remaining extended situated literal of the form '*not* $R{:}l$';
- each rule with a situated literal '$R{:}l$' that is not Q-local such that $R{:}l \notin I$;
- each remaining situated literal $R{:}l$ that is not Q-local.

The underlying intuition of the reduct is clear. Analogous to the definition of the reduct for normal programs [15], the reduct of a communicating normal program defines a way to reduce a program relative to some guess I. The reduct of a communicating normal program is a communicating simple program that only contains component simple programs Q with Q-local situated literals. That is, each component simple program Q corresponds to a classical simple program.

Definition 1. *We say that an interpretation I of a communicating normal program $\mathcal{P}$ is an* answer set *of $\mathcal{P}$ if and only if $\forall Q \in \mathcal{P} \cdot (Q{:}I_{\downarrow Q}) = \left(Q^I\right)^\star$.*

In other words: an interpretation I is an answer set of a communicating normal program $\mathcal{P}$ if and only if for every component normal program Q we have that the projection of I on Q is an answer set of the component normal program Q under the classical definition.

Example 3. Let us once again consider the communicating simple program $\mathcal{P} = \{Q, R\}$ from Example 2. Given the interpretation $I = \{Q{:}a, Q{:}b, R{:}a\}$ we find that $Q^I = \{a \leftarrow, b \leftarrow\}$ and $R^I = \{a \leftarrow\}$. We can easily treat Q^I and R^I separately since they now correspond to classical programs. It then readily follows that $Q{:}I_{\downarrow Q} = \left(Q^I\right)^\star$ and $R{:}I_{\downarrow R} = \left(R^I\right)^\star$, hence the interpretation I is an answer set of $\mathcal{P}$. In total the communicating simple program $\mathcal{P}$ has two answer sets, namely $\{Q{:}b\}$ and $\{Q{:}a, Q{:}b, R{:}a\}$.

Proposition 1. *[1] Let $\mathcal{P}$ be a communicating simple program. Determining whether a given situated literal $Q{:}l$ occurs in any answer set of $\mathcal{P}$ (*i.e.* brave reasoning) is in* NP*. Determining whether a literal is true in all the answer sets of $\mathcal{P}$ (*i.e.* cautious reasoning) is in* coNP*. Furthermore, the same complexity results hold when $\mathcal{P}$ is a communicating normal program.*

4 Multi-focused Answer Sets

We extend the semantics of communicating programs in such a way that it becomes possible to focus on a sequence of component programs. As such, we can indicate that we are only interested in those answer sets that are successively minimal with respect to each respective component program. The underlying intuition is that of leaders and followers, where the decisions that an agent can make are limited by what its leaders have previously decided.

Definition 2. *Let $\mathcal{P}$ be a communicating program and $\{Q_1, \ldots, Q_n\} \subseteq \mathcal{P}$ a set of component programs. A $(Q_1, \ldots, Q_n)$*-focused answer set *of $\mathcal{P}$ is defined as:*

$$\begin{aligned}&M \textit{ is a } (Q_1, \ldots, Q_{n-1})\textit{-focused answer set of } \mathcal{P} \textit{ and}\\ &\quad \textit{for each } (Q_1, \ldots, Q_{n-1})\textit{-focused answer set } M' \textit{ of } \mathcal{P}\\ &\quad \textit{we do not have that } M'_{\downarrow Q_n} \subset M_{\downarrow Q_n}\end{aligned}$$

where a ()-focused answer set of $\mathcal{P}$ is any answer set of $\mathcal{P}$.

In other words, we say that M is a $(Q_1, \ldots, Q_n)$-focused answer set of $\mathcal{P}$ if and only if M is minimal among all $(Q_1, \ldots, Q_{n-1})$-focused answer sets *w.r.t.* the projection on Q_n.

Example 4. Consider the communicating program $\mathcal{P}_4 = \{Q, R, S\}$ with the rules

$$\begin{array}{lll} Q{:}a \leftarrow & R{:}b \leftarrow S{:}c & S{:}a \leftarrow \\ Q{:}b \leftarrow \textit{not } S{:}d & R{:}a \leftarrow S{:}c & S{:}c \leftarrow \textit{not } S{:}d, \textit{not } R{:}c \\ Q{:}c \leftarrow R{:}c & R{:}a \leftarrow S{:}d & S{:}c \leftarrow \textit{not } S{:}c, \textit{not } R{:}c \\ & R{:}c \leftarrow \textit{not } R{:}a & \end{array}$$

The communicating program $\mathcal{P}_4$ has three answer sets, namely

$$\begin{aligned} M_1 &= Q{:}\{a,b,c\} \cup R{:}\{c\} \cup S{:}\{a\} \\ M_2 &= Q{:}\{a,b\} \cup R{:}\{a,b\} \cup S{:}\{a,c\} \\ M_3 &= Q{:}\{a\} \cup R{:}\{a\} \cup S{:}\{a,d\}. \end{aligned}$$

The only (R, S)-focused answer set of $\mathcal{P}_4$ is M_1. Indeed, since $\{a\} = (M_3)_{\downarrow R} \subset (M_2)_{\downarrow R} = \{a, b\}$ we find that M_2 is not a (R)-focused answer set. Furthermore $\{a\} = (M_1)_{\downarrow S} \subset (M_3)_{\downarrow S} = \{a, d\}$, hence M_3 is not an (R, S)-focused answer set.

We now show how the validity of quantified boolean formulas (QBF) can be checked using multi-focused answer sets of communicating ASP programs.

Definition 3. *Let $\phi = \exists X_1 \forall X_2 \ldots \Theta X_n \cdot p(X_1, X_2, \cdots X_n)$ be a QBF where $\Theta = \forall$ if n is even and $\Theta = \exists$ otherwise, and $p(X_1, X_2, \cdots X_n)$ is a formula of the form $\theta_1 \vee \ldots \vee \theta_m$ in disjunctive normal form over $X_1 \cup \ldots \cup X_n$ with X_i, $1 \leq i \leq n$, sets of variables and where each θ_t is a conjunction of propositional literals. We define Q_0 as follows:*

$$\begin{aligned} Q_0 = &\{x \leftarrow \textit{not } \neg x, \neg x \leftarrow \textit{not } x \mid x \in X_1 \cup \ldots \cup X_n\} && (6)\\ &\cup \{\textit{sat} \leftarrow Q_0{:}\theta_t \mid \theta_t, 1 \leq t \leq m\} && (7)\\ &\cup \{\neg \textit{sat} \leftarrow \textit{not } \textit{sat}\}\,. && (8) \end{aligned}$$

For $1 \leq j \leq n-1$ we define Q_j as follows:

$$\begin{aligned} Q_j = &\{x \leftarrow Q_0{:}x, \neg x \leftarrow Q_0{:}\neg x \mid x \in (X_1 \cup \ldots \cup X_{n-j})\} && (9)\\ &\cup \begin{cases} \{\neg \textit{sat} \leftarrow Q_0{:}\neg \textit{sat}\} & \textit{if } (n-j) \textit{ is even} \\ \{\textit{sat} \leftarrow Q_0{:}\textit{sat}\} & \textit{if } (n-j) \textit{ is odd.} \end{cases} && (10) \end{aligned}$$

The communicating normal program corresponding with ϕ is $\mathcal{P} = \{Q_0, ..., Q_{n-1}\}$.

For a QBF of the form $\phi = \forall X_1 \exists X_2 ... \Theta X_n \cdot p(X_1, X_2, \cdots X_n)$ where $\Theta = \exists$ if n is even and $\Theta = \forall$ otherwise and $p(X_1, X_2, \cdots X_n)$ once again a formula in disjunctive normal form, the simulation only changes slightly. Indeed, only the conditions in (10) are swapped.

Example 5. Given the QBF $\phi = \exists x \forall y \exists z \cdot (x \wedge y) \vee (\neg x \wedge y \wedge z) \vee (\neg x \wedge \neg y \wedge \neg z)$, the communicating program $\mathcal{P}$ corresponding with the QBF ϕ is defined as follows:

$$\begin{array}{ccc}
Q_0{:}x \leftarrow not\ \neg x & Q_0{:}y \leftarrow not\ \neg y & Q_0{:}z \leftarrow not\ \neg z \\
Q_0{:}\neg x \leftarrow not\ x & Q_0{:}\neg y \leftarrow not\ y & Q_0{:}\neg z \leftarrow not\ z \\
Q_0{:}sat \leftarrow x, y & Q_0{:}sat \leftarrow \neg x, y, z & Q_0{:}sat \leftarrow \neg x, \neg y, \neg z \\
Q_0{:}\neg sat \leftarrow not\ sat & &
\end{array}$$

$$\begin{array}{ccc}
Q_1{:}x \leftarrow Q_0{:}x & Q_1{:}y \leftarrow Q_0{:}y & \\
Q_1{:}\neg x \leftarrow Q_0{:}\neg x & Q_1{:}\neg y \leftarrow Q_0{:}\neg y & Q_1{:}\neg sat \leftarrow Q_0{:}\neg sat
\end{array}$$

$$\begin{array}{ccc}
Q_2{:}x \leftarrow Q_0{:}x & Q_2{:}\neg x \leftarrow Q_0{:}\neg x & Q_2{:}sat \leftarrow Q_0{:}sat
\end{array}$$

The communicating program in Example 5 can be used to determine whether the QBF ϕ is satisfiable. First, note that the rules in (6) generate all possible truth assignments of the variables, *i.e.* all possible propositional interpretations. The rules in (7) ensure that '*sat*' is true exactly for those interpretations that satisfy the formula $p(X_1, X_2, \ldots, X_n)$, *i.e.* we check, for this particular assignment of the variables, whether $p(X_1, \ldots, X_n)$ is satisfied.

Intuitively, the component programs $\{Q_1, \ldots, Q_{n-1}\}$ successively bind fewer and fewer variables. In particular, focusing on $Q_1, \ldots, Q_{n-1}$ allows us to consider the binding of the variables in $X_{n-1}, \ldots, X_1$, respectively. Depending on the rules from (10), focusing on Q_i allows us to verify that either some or all of the assignments of the variables in X_{n-j} make the formula $p(X_1, \ldots, X_n)$ satisfied, given the bindings that have already been determined by the preceding components. We now prove that the QBF ϕ is satisfiable iff $Q_0{:}sat$ is true in some $(Q_1, \ldots, Q_{n-1})$-focused answer set.

Proposition 2. *Let ϕ and $\mathcal{P}$ be as in Definition 3. We have that a QBF ϕ of the form $\phi = \exists X_1 \forall X_2 ... \Theta X_n \cdot p(X_1, X_2, \cdots X_n)$ is satisfiable if and only if $Q_0{:}sat$ is true in some $(Q_1, \ldots, Q_{n-1})$-focused answer set of $\mathcal{P}$. Furthermore, we have that a QBF ϕ of the form $\phi = \forall X_1 \exists X_2 ... \Theta X_n \cdot p(X_1, X_2, \cdots X_n)$ is satisfiable if and only if $Q_0{:}sat$ is true in all $(Q_1, \ldots, Q_{n-1})$-focused answer sets of $\mathcal{P}$.*

Proof. We give a proof by induction. Assume we have a QBF ϕ_1 of the form $\exists X_1 \cdot p(X_1)$ with $\mathcal{P}_1 = \{Q_0\}$ the communicating normal program corresponding with ϕ_1 according to Definition 3. If the formula $p_1(X_1)$ of the QBF ϕ_1 is satisfiable then we know that there is a ()-focused answer set M of $\mathcal{P}_1$ such that $Q_0{:}sat \in M$. Otherwise, we know that $Q_0{:}sat \notin M$ for all ()-answer sets M of $\mathcal{P}_1$. Hence the induction hypothesis is valid for $n = 1$.

Assume the result holds for any QBF ϕ_{n-1} of the form $\exists X_1 \forall X_2 \ldots \Theta X_{n-1} \cdot p_n(X_1, X_2, \ldots, X_{n-1})$. We show in the induction step that it holds for any QBF ϕ_n of the form $\exists X_1 \forall X_2 \ldots \overline{\Theta} X_n \cdot p_{n-1}(X_1, X_2, \ldots, X_n)$. Let $\mathcal{P} = \{Q_0, \ldots, Q_{n-1}\}$ and $\mathcal{P}' = \{Q'_0, \ldots, Q'_{n-2}\}$ be the communicating normal programs that correspond with ϕ_n and ϕ_{n-1}, respectively. Note that the component programs $Q_2, \ldots, Q_{n-1}$ are defined in exactly the same way as the component programs $Q'_1, \ldots, Q'_{n-2}$, the only difference being the name of the component programs. What is of importance in the case of ϕ_n is therefore only the additional rules in Q_0 and the new component program Q_1. The additional rules in Q_0 merely generate the corresponding interpretations, where we now need to consider the possible interpretations of the variables from X_n as well. The rules in the new component program Q_1 ensure that $Q_1 {:} x \in M$ whenever $Q_0 {:} x \in M$ and $Q_1 {:} \neg x \in M$ whenever $Q_0 {:} \neg x \in M$ for every M an answer set of $\mathcal{P}$ and $x \in (X_1 \cup \ldots \cup X_{n-1})$. Depending on n being even or odd, we get two distinct cases:

- if n is even, then we have $(sat \leftarrow Q_0 {:} sat) \in Q_1$ and we know that the QBF ϕ_n has the form $\exists X_1 \forall X_2 \ldots \forall X_n \cdot p_n(X_1, X_2, \ldots, X_n)$. Let us consider what happens when we determine the (Q_1)-focused answer sets of $\mathcal{P}$. Due to the construction of Q_1, we know that $M'_{\downarrow Q_1} \subset M_{\downarrow Q_1}$ can only hold for two answer sets M' and M of $\mathcal{P}$ if M' and M correspond to identical interpretations of the variables in $X_1 \cup \ldots \cup X_{n-1}$. Furthermore, $M'_{\downarrow Q_1} \subset M_{\downarrow Q_1}$ is only possible if $Q_1 {:} sat \in M$ while $Q_1 {:} sat \notin M'$.

  Now note that given an interpretation of the variables in $X_1 \cup \ldots \cup X_{n-1}$, there is exactly one answer set for each choice of X_n. When we have M' with $Q_1 {:} sat \notin M'$ this implies that there is an interpretation such that, for some choice of X_n, this particular assignment of values of the QBF does not satisfy the QBF. Similarly, if we have M with $Q_1 {:} sat \in M$ then the QBF is satisfied for that particular choice of X_n. Determining (Q_1)-focused answer sets of $\mathcal{P}$ will eliminate M since $M'_{\downarrow Q_1} \subset M_{\downarrow Q_1}$. In other words, for identical interpretations of the variables in $X_1 \cup \ldots \cup X_{n-1}$, the answer set M' encodes a counterexample that shows that for these interpretations it does not hold that the QBF is satisfied for all choices of X_n. Focusing thus eliminates those answer sets that claim that the QBF is satisfiable for the variables in $X_1 \cup \ldots \cup X_{n-1}$. When we cannot find such $M'_{\downarrow Q_1} \subset M_{\downarrow Q_1}$ this is either because none of the interpretations satisfy the QBF or all of the interpretations satisfy the QBF. In both cases, there is no need to eliminate any answer sets. We thus effectively mimic the requirement that the QBF ϕ_n should hold for $\forall X_n$.

- if n is odd, then $(\neg sat \leftarrow Q_0 {:} \neg sat) \in Q_1$ and we know that the QBF ϕ_n has the form $\exists X_1 \forall X_2 \ldots \exists X_n \cdot p_n(X_1, X_2, \ldots, X_n)$. As before, we know that $M'_{\downarrow Q_1} \subset M_{\downarrow Q_1}$ can only hold for two answer sets M' and M of $\mathcal{P}$ if M' and M correspond to identical interpretations of the variables in $X_1 \cup \ldots \cup X_{n-1}$. However, this time $M'_{\downarrow Q_1} \subset M_{\downarrow Q_1}$ is only possible if $Q_1 {:} \neg sat \in M$ while $Q_1 {:} \neg sat \notin M'$.

  If we have M with $Q_1 {:} \neg sat \in M$ then the QBF is not satisfied for that particular choice of X_n, whereas when M' with $Q_1 {:} \neg sat \notin M'$ this implies that there is an interpretation such that, for some choice of X_n, this particular assignment of the variables does satisfy the QBF. Determining (Q_1)-focused answer sets of $\mathcal{P}$ will eliminate M since $M'_{\downarrow Q_1} \subset M_{\downarrow Q_1}$. For identical interpretations of the variables in $X_1 \cup \ldots \cup X_{n-1}$, the answer set M' encodes a counterexample that shows that for these interpretations there is some choice of X_n such that the QBF is satisfied. Focusing thus eliminates those answer sets that claim that the QBF is not satisfiable for the variables in $X_1 \cup \ldots \cup X_{n-1}$. When we cannot find such $M'_{\downarrow Q_1} \subset M_{\downarrow Q_1}$ this is either because none of the interpretations satisfy the QBF or all of the interpretations satisfy the QBF. In both cases, there is no need to eliminate any answer sets. We effectively mimic the requirement that the QBF ϕ_n should hold for $\exists X_n$.

For a QBF of the form $\forall X_1 \exists X_2 \ldots \Theta X_n \cdot p(X_1, X_2, \ldots, X_n)$, with $\Theta = \exists$ if n is even and $\Theta = \forall$ otherwise, the proof is analogous. In the base case, we know that a QBF ϕ_1 of the form $\forall X_1 \cdot p(X_1)$ is satisfiable only when for every ()-focused answer set M of $\mathcal{P}_1 = \{Q_0\}$ we find that $Q_0 : sat \in M$. Otherwise, we know that there exists some ()-focused answers sets M of $\mathcal{P}_1$ such that $Q_0 : sat \notin M$. Hence the induction hypothesis is valid for $n = 1$. The induction step is then entirely analogous to what we have proven before, with the only difference being that the cases for n being even or odd are swapped. Finally, since the first quantifier is $\forall$, we need to verify that $Q_0 : sat$ is true in every $(Q_1, \ldots, Q_{n-1})$-focused answer set of $\mathcal{P}$. □

Before we discuss the computational complexity, we recall some of the notions of complexity theory. We have $\Sigma_0^{\mathsf{P}} = \mathsf{P}$ and $\Sigma_1^{\mathsf{P}} = \mathsf{NP}$ where the complexity class $\Sigma_n^{\mathsf{P}} = \mathsf{NP}^{\Sigma_{n-1}^{\mathsf{P}}}$ is the class of problems that can be solved in polynomial time on a non-deterministic machine with an $\Sigma_{n-1}^{\mathsf{P}}$ oracle, *i.e.* assuming a procedure that can solve $\Sigma_{n-1}^{\mathsf{P}}$ problems in constant time [19]. Deciding the validity of a QBF $\phi = \exists X_1 \forall X_2 ... \Theta X_n \cdot p(X_1, X_2, \cdots X_n)$ is the canonical Σ_n^{P}-complete problem. Furthermore $\Pi_n^{\mathsf{P}} = \mathsf{co}(\Sigma_n^{\mathsf{P}})$. Deciding the validity of a QBF $\phi = \forall X_1 \exists X_2 ... \Theta X_n \cdot p(X_1, X_2, \cdots X_n)$ with $\Theta = \forall$ if n is odd and $\Theta = \exists$ otherwise, is the canonical Π_n^{P}-complete problem.

Corollary 1. *Let $\mathcal{P}$ be a communicating program, $Q_i \in \mathcal{P}$. The problem of deciding whether $Q_i : l \in M$ (brave reasoning) with M a $(Q_1, \ldots, Q_n)$-focused answer set of $\mathcal{P}$ is $\Sigma_{n+1}^{\mathsf{P}}$-hard.*

Corollary 2. *Let $\mathcal{P}$ be a communicating program, $Q_i \in \mathcal{P}$. The problem of deciding whether all $(Q_1, \ldots, Q_n)$-focused answer sets contain $Q_i : l$ (cautious reasoning) is Π_{n+1}^{P}-hard.*

In addition to these hardness results, we can also establish the corresponding membership results.

Proposition 3. *Let $\mathcal{P}$ be a communicating program, $Q_i \in \mathcal{P}$. Deciding whether $Q_i : l \in M$ with M a $(Q_1, \ldots, Q_n)$-focused answer set of $\mathcal{P}$ is in $\Sigma_{n+1}^{\mathsf{P}}$.*

Proof. (sketch) This proof is by induction on n. When $n = 1$ we guess a (Q_1)-focused answer set M of $\mathcal{P}$ in polynomial time and verify that this is indeed a (Q_1)-focused answer set in coNP as in classical ASP, *i.e.* finding a (Q_1)-focused answer set is in Σ_2^{P}. Given an algorithm to compute the $(Q_1, \ldots, Q_{n-1})$-focused answer sets of $\mathcal{P}$ in Σ_n^{P}, we can guess a $(Q_1, \ldots, Q_n)$-focused answer set and verify there is no $(Q_1, \ldots, Q_n)$-focused answer set M' of $\mathcal{P}$ such that $M'_{\downarrow Q_n} \subset M_{\downarrow Q_n}$ using a Σ_n^{P} oracle, *i.e.* the algorithm is in $\Sigma_{n+1}^{\mathsf{P}}$. $\square$

Now that we have both hardness and membership results, we readily obtain the following corollary.

Corollary 3. *Let $\mathcal{P}$ be a communicating normal program, $Q_i \in \mathcal{P}$. The problem of deciding whether $Q_i\!:\!l \in M$ with M a $(Q_1, \ldots, Q_n)$-focused answer set of $\mathcal{P}$ is $\Sigma_{n+1}^{\mathsf{P}}$-complete.*

The next corollary provides a result for communicating simple programs instead of communicating normal programs.

Corollary 4. *Let $\mathcal{P}$ be a communicating simple program, $Q_i \in \mathcal{P}$. The problem of deciding whether $Q_i\!:\!l \in M$ with M a $(Q_1, \ldots, Q_n)$-focused answer set of $\mathcal{P}$ is $\Sigma_{n+1}^{\mathsf{P}}$-complete.*

5 Related Work

A lot of research has been done, for various reasons, on the subject of combining logic programming with the multi-agent paradigm. One reason for such a combination is that logics can be used to describe the (rational) behavior of the agents [9]. Another reason is that it can be used to combine different flavors of logic programming languages [11,18]. Such an extension of logic programming can be used to externally solve tasks for which ASP is not suited, while remaining in a declarative framework [13]. It can also be used as a form of cooperation, where multiple agents or contexts collaborate to solve a difficult problem [8,21]. The approach in this paper falls in the last category and is concerned with how the collaboration of different ASP programs affect the expressiveness of the overall system.

Important work has been done in the domain of multi-context systems (MCS) and multi-agent ASP to enable collaboration between the different contexts/ASP programs. We briefly discuss some of the more prominent work in these areas.

The work of [20] discusses an extension of MCSs [16] that allows MCSs to reason about absent information. Each context only has access to a subset of the available information. In order to share this information, an information flow is defined by the system between the different contexts. This idea was later adopted in the ASP community and in our work in particular.

The current paper has the same syntax as [20] but rather different semantics. The semantics in [20] are closely related to the well-founded semantics [14], whereas ours are closer to the spirit of stable models [15]. Another point where our semantics differ is that the motivation for accepting a literal as being true can be circular if that explanation relies on other component programs. In [5] this circularity is identified as a requirement for the representation of social reasoning.

The work in [4] further extends the work in [20], introducing a multi-context variant of default logic. This extension guarantees that a number of conclusions from default logic come "for free". The paper is the first to offer a syntactical description of the communication rather than a semantic one, making it easier to implement an actual algorithm. Some of interesting applications of contextual frameworks are shown, *e.g.* information fusion, game theory and social choice.

Along similar lines [3] combines the non-monotonicity from [20] with the heterogeneous approach from [16] into a single framework for heterogenous non-monotonic multi-context reasoning. The work in [3] introduces several notions of equilibria, including minimal and grounded equilibria. In our approach, local reasoning is captured by grounded equilibria (no circularity allowed) while communicating with other component programs is captured by the weaker concept of minimal equilibria. The work in [3] also offers various membership results on checking the existence of an equilibrium. Most notably, [3] is – to the best of our knowledge – the first to explicitly remark that multi-context systems can be non-monotonic even if all the logics in the component programs are monotonic.

We now direct our attention to work done within the ASP community. The ideas presented in this paper are related to HEX programs [12] in which ASP is extended by higher-order predicates and external atoms. Through these external atoms, knowledge can be exchanged with external sources while remaining within the declarative paradigm. Applicability-wise, HEX is proposed as a tool for non-monotonic semantic web reasoning under the answer set semantics. Because of this setting, HEX is not primarily targeted at increasing the expressiveness, but foremost at extending the applicability and ease of use of ASP.

Two other important works in the area of multi-agent ASP are [8] and [21]. In both [8] and [21] a multi-agent system is developed in which multiple agents communicate with each other. The communication channel is uni-directional, allowing information to be pushed to the next agent. Both approaches use ASP and have agents that are quite expressive in their own right. Indeed, in [8] each agent is an Ordered Choice Logic Program [2] and in [21] each agent uses the extended answer set semantics.

In [8] the agents can communicate with whomever they want and circular communication is allowed (agent A tells B whom tells A ...), which is similar to our approach. However, in [8] only positive information can be shared and the authors do not look at the actual expressiveness of the framework. In [21] Hierarchical Decision Making is introduced where each agent uses the extended answer set semantics. The network is a linear "hierarchical" network and the framework employs the idea of a failure feedback mechanism. Intuitively, this mechanism allows the previous agent in a network to revise his conclusion when it leads to an unresolvable inconsistency for the next agent in the network. It is this mechanism that gives rise to a higher expressiveness, namely Σ_n^P for a hierarchical network of n agents. Our work is different in that we start from normal and simple ASP programs for the agents. Our communication mechanism is also quite simple and does not rely on any kind of feedback. Regardless, we obtain a comparable expressiveness.

We briefly mention [7], in which recursive modular non-monotonic logic programs (MLP) under the ASP semantics are considered. The main difference between MLP and our simple communication is that our communication is parameter-less, *i.e.* the truth of a situated literal is not dependent on parameters passed by the situated literal to the target component program. Our approach is clearly different and we cannot readily mimic the behavior of the network as presented in [7]. Our complexity results therefore do not directly apply to MLPs.

Finally, we like to point out the resemblance between multi-focused answer sets and the work on multi-level linear programming [17]. In multi-level linear programming, different agents control different variables that are outside of the control of the other agents, yet are linked by means of linear inequalities (constraints). The agents have to fix the values of the variables they can control in a predefined order, such that their own linear objective function is optimized. Similarly, in communicating ASP, literals belong to different component programs (agents), and their values are linked through constraints, which in this case take the form of rules. Again the agents act in a predefined order, but now they try to minimize the set of literals they have to accept as being true, rather than a linear objective function.

6 Conclusion

We have introduced multi-focused answer sets for communicating programs. The underlying intuition is that of leaders and followers, where the choices available to the followers are limited by what the leaders have previously decided. On a technical level, the problem translates to establishing local minimality for some of the component programs in the communicating program. Since in general it is not possible to ensure local minimality for all component programs, an order must be defined among component programs on which to focus. The result is an increase in expressiveness, where the problem of deciding whether $Q_i : l \in M$ with M a $(Q_1, \ldots, Q_n)$-focused answer set of $\mathcal{P}$ is $\Sigma^{\mathsf{P}}_{n+1}$-complete. In our work we thus find that the choice of the communication mechanism is paramount *w.r.t.* the expressiveness of the overall system, irrespective of the expressiveness of the individual agents.

References

1. Bauters, K., Janssen, J., Schockaert, S., Vermeir, D., De Cock, M.: Communicating answer set programs. In: Tech. Comm. of ICLP 2010, vol. 7, pp. 34–43 (2010)
2. Brain, M., De Vos, M.: Implementing OCLP as a front-end for answer set solvers: From theory to practice. In: Proc. of ASP 2005 (2003)
3. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proc. of AAAI 2007, pp. 385–390 (2007)

4. Brewka, G., Roelofsen, F., Serafini, L.: Contextual default reasoning. In: Proc. of. IJCAI 2007, pp. 268–273 (2007)
5. Buccafurri, F., Caminiti, G., Laurendi, R.: A logic language with stable model semantics for social reasoning. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 718–723. Springer, Heidelberg (2008)
6. Bylander, T.: The computational complexity of propositional STRIPS planning. Artificial Intelligence 69, 165–204 (1994)
7. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular nonmonotonic logic programming revisited. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 145–159. Springer, Heidelberg (2009)
8. De Vos, M., Crick, T., Padget, J., Brain, M., Cliffe, O., Needham, J.: LAIMA: A multi-agent platform using ordered choice logic programming. In: Baldoni, M., Endriss, U., Zhang, S.-W., Torroni, P. (eds.) DALT 2005. LNCS (LNAI), vol. 3904, pp. 72–88. Springer, Heidelberg (2006)
9. Dell'Acqua, P., Sadri, F., Toni, F.: Communicating agents. In: Proc. of MASL 1999 (1999)
10. Eiter, T., Gottlob, G.: The complexity of logic-based abduction. Journal of the ACM 42, 3–42 (1995)
11. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artifial Intelligence 172(12-13), 1495–1539 (2008)
12. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Proc. of IJCAI 2005, pp. 90–96 (2005)
13. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: dlvhex: A tool for semantic-web reasoning under the answer-set semantics. In: Proc. of ALPSWS 2006, pp. 33–39 (2006)
14. Gelder, A.V., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)
15. Gelfond, M., Lifzchitz, V.: The stable model semantics for logic programming. In: Proc. of ICLP 1988, pp. 1081–1086 (1988)
16. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics or: How we can do without modal logics. Artifial Intelligence 65(1), 29–70 (1994)
17. Jeroslow, R.: The polynomial hierarchy and a simple model for competitive analysis. Mathematical Programming 32, 146–164 (1985)
18. Luo, J., Shi, Z., Wang, M., Huang, H.: Multi-agent cooperation: A description logic view. In: Lukose, D., Shi, Z. (eds.) PRIMA 2005. LNCS, vol. 4078, pp. 365–379. Springer, Heidelberg (2009)
19. Papadimitriou, C.: Computational complexity. Addison-Wesley, Reading (1994)
20. Roelofsen, F., Serafini, L.: Minimal and absent information in contexts. In: Proc. of IJCAI 2005, pp. 558–563 (2005)
21. Van Nieuwenborgh, D., De Vos, M., Heymans, S., Hadavandi, E.: Hierarchical decision making in multi-agent systems using answer set programming. In: Inoue, K., Satoh, K., Toni, F. (eds.) CLIMA 2006. LNCS (LNAI), vol. 4371, pp. 20–40. Springer, Heidelberg (2007)

Loop Formulas for Splitable Temporal Logic Programs*

Felicidad Aguado, Pedro Cabalar, Gilberto Pérez, and Concepción Vidal

Department of Computer Science,
University of Corunna (Spain)
{aguado,cabalar,gperez,eicovima}@udc.es

Abstract. In this paper, we study a method for computing *temporal equilibrium models*, a generalisation of stable models for logic programs with temporal operators, as in Linear Temporal Logic (LTL). To this aim, we focus on a syntactic subclass of these temporal logic programs called *splitable* and whose main property is satisfying a kind of "future projected" dependence present in most dynamic scenarios in Answer Set Programming (ASP). Informally speaking, this property can be expressed as "past does not depend on the future." We show that for this syntactic class, temporal equilibrium models can be captured by an LTL formula, that results from the combination of two well-known techniques in ASP: splitting and loop formulas. As a result, an LTL model checker can be used to obtain the temporal equilibrium models of the program.

1 Introduction

Although transition systems frequently appear in scenarios and applications of Non-Monotonic Reasoning (NMR), most NMR formalisms are not particularly thought for temporal reasoning. Instead, NMR approaches are typically "static," in the sense that time instants are treated as one more argument for predicates representing actions and fluents. This has been usual, for instance, when representing temporal scenarios in Answer Set Programming (ASP) [1,2], a successful NMR paradigm based on the *stable model* semantics [3] for logic programs. In this case, it is frequent that program rules depend on a parameter T, the previous situation, and the value $T{+}1$ representing its successor state. For instance, if t represents the action of toggling a switch that can take two positions, d (down) and u (up), the corresponding *effect axioms* would be encoded as:

$$u(T{+}1) \leftarrow t(T), d(T) \tag{1}$$
$$d(T{+}1) \leftarrow t(T), u(T) \tag{2}$$

Similarly, the *inertia law* would typically look like the pair of rules:

$$u(T{+}1) \leftarrow u(T), \mathit{not}\ d(T{+}1) \tag{3}$$
$$d(T{+}1) \leftarrow d(T), \mathit{not}\ u(T{+}1) \tag{4}$$

* This research was partially supported by Spanish MEC project TIN2009-14562-C05-04 and Xunta de Galicia project INCITE08-PXIB105159PR.

Since ASP tools are constrained to finite domains, a finite bound n for the number of transitions is usually fixed, so that the above rules are grounded for $T = 0, \dots, n-1$. To solve a planning problem, for instance, we would iterate multiple calls to some ASP solver and go increasing the value of $n = 1, 2, 3, \dots$ in each call, until a (minimal length) plan is found.

Of course, this strategy falls short for many temporal reasoning problems, like proving the non-existence of a plan, or checking whether two NMR system representations are *strongly equivalent*, that is, whether they always have the same behaviour, even after adding some common piece of knowledge, and *for any narrative length* we consider.

To overcome these limitations, [4] introduced an extension of ASP to deal with modal temporal operators. Such an extension was the result of mixing two logical formalisms: (1) *Equilibrium Logic* [5,6] that widens the concept of stable models to the general syntax of arbitrary theories (propositional and even first order); and (2) the well-known *Linear Temporal Logic* [7] (LTL) dealing with operators like $\square$ (read as "always"), $\Diamond$ ("eventually"), $\bigcirc$ ("next"), $\mathcal{U}$ ("until") and $\mathcal{R}$ ("release"). The result of this combination received the name of *Temporal Equilibrium Logic* (TEL). As happens with Equilibrium Logic, TEL is defined in terms of a monotonic formalism, in this case called *Temporal Here-and-There* (THT), plus an ordering relation among models, so that only the minimal ones are selected (inducing a non-monotonic consequence relation). These minimal models receive the name of *Temporal Equilibrium Models* and can be seen as the analogous of *stable models* for the temporal case. As an example, the rules (1)-(4) can be respectively encoded in TEL as:

$$\square(d \wedge t \rightarrow \bigcirc u) \tag{5}$$

$$\square(u \wedge t \rightarrow \bigcirc d) \tag{6}$$

$$\square(u \wedge \neg\bigcirc d \rightarrow \bigcirc u) \tag{7}$$

$$\square(d \wedge \neg\bigcirc u \rightarrow \bigcirc d) \tag{8}$$

In [8] it was shown how to use equivalence in the logic of THT as a sufficient condition for strong equivalence of two arbitrary TEL theories. The THT-equivalence test was performed by translating THT into LTL and using a model checker afterwards. This technique was applied on several examples to compare different alternative ASP representations of the same temporal scenario. Paradoxically, although this allowed an automated test to know whether two representations had the same behaviour, computing such a behaviour, that is, automatically obtaining the temporal equilibrium models of a given theory was still an open topic.

When compared to the ASP case, part of the difficulties for computing temporal equilibrium models came from the fact that the general syntax of TEL allows complete arbitrary nesting of connectives (that is, it coincides with general LTL), whereas ASP syntax is constrained to disjunctive logic programs, that is, rules with a *body* (the antecedent) with a conjunction of literals, and a *head* (the consequent) with a disjunction of atoms. In a recent work [9], it was shown that TEL could be reduced to a normal form closer to logic programs where, roughly speaking, in place of an atom p we can also use $\bigcirc p$, and any rule can be embraced by $\Box$. This normal form received the name of *Temporal Logic Programs* (TLPs) – for instance, (5)-(8) are TLP rules.

In this work we show how to compute the temporal equilibrium models for a subclass of TLPs called *splitable*. This subclass has a property we can call *future projected dependence* and that informally speaking can be described as "past does not depend on the future." Formally, this means that we cannot have rules where a head atom without $\bigcirc$ depends on a body atom with $\bigcirc$. In our example, (5)-(8) are also splitable TLP rules whereas, for instance, the following TLP rules are not in splitable form:

$$\Box(\neg\bigcirc p \rightarrow p) \tag{9}$$

$$\Box(\bigcirc p \rightarrow p) \tag{10}$$

since the truth of p "now" depends on p in the next situation $\bigcirc p$. This syntactic feature of splitable TLPs allows us to apply the so-called *splitting* technique [10] (hence the name of *splitable*) to our temporal programs. Informally speaking, splitting is applicable when we can divide an ASP program Π into a bottom Π_0 and a top Π_1 part, where Π_0 never depends on predicates defined in Π_1. If so, the stable models of Π can be computed by first obtaining the stable models of Π_0, and then using them to simplify Π_1 and compute the rest of information in a constructive way.

In the case of splitable TLPs, however, we cannot apply splitting by relying of multiple calls to an ASP solver since, rather than a single split between bottom and top part, we would actually have an *infinite* sequence of program "slices" $\Pi_0, \Pi_1, \Pi_2, \ldots$ where each Π_i depends on the previous ones. To solve this problem, we adopt a second ASP technique called *Loop Formulas* [11,12], a set of formulas $LF(\Pi)$ for some program Π so that the stable models of the latter coincide with the classical models of $\Pi \cup LF(\Pi)$. In our case, $LF(\Pi)$ will contain formulas preceded by a $\Box$ operator, so that they affect to *all* program slices. As a result, the temporal equilibrium models of Π will correspond to the LTL models of $\Pi \cup LF(\Pi)$, which can be computed using an LTL model checker as a back-end.

The rest of the paper is organised as follows. In the next section, we recall the syntax and semantics of TEL. Section 3 describes the syntax of splitable TLPs and their relation to stable models. Next, in Section 5, we explain how to construct the set of loop formulas $LF(\Pi)$ for any splitable TLP Π, proving also that the LTL models of $\Pi \cup LF(\Pi)$ are the temporal equilibrium models of Π. Finally, Section 6 concludes the paper.

2 Preliminaries

Given a set of atoms At, a *formula* F is defined as in LTL following the grammar:

$$F ::= p \mid \bot \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid \bigcirc F \mid \Box F \mid \Diamond F$$

where $p \in At$. A *theory* is a finite set of formulas. We use the following derived operators[1] and notation:

$$\neg F \overset{\text{def}}{=} F \to \bot \qquad \bigcirc^0 F \overset{\text{def}}{=} F$$
$$\top \overset{\text{def}}{=} \neg\bot \qquad \bigcirc^i F \overset{\text{def}}{=} \bigcirc(\bigcirc^{i-1} F) \quad (\text{with } i > 1)$$
$$F \leftrightarrow G \overset{\text{def}}{=} (F \to G) \wedge (G \to F) \qquad \Gamma^\vee \overset{\text{def}}{=} \bigvee_{F \in \Gamma} F$$

for any formulas F, G and set of formulas Γ.

The semantics of the logic of *Temporal Here-and-There* (THT) is defined in terms of sequences of pairs of propositional interpretations. A (temporal) *interpretation* $\mathbf{M}$ is an infinite sequence of pairs $m_i = \langle H_i, T_i \rangle$ with $i = 0, 1, 2, \dots$ where $H_i \subseteq T_i$ are sets of atoms standing for *here* and *there* respectively. For simplicity, given a temporal interpretation, we write $\mathbf{H}$ (resp. $\mathbf{T}$) to denote the sequence of pair components $H_0, H_1, \dots$ (resp. $T_0, T_1, \dots$). Using this notation, we will sometimes abbreviate the interpretation as $\mathbf{M} = \langle \mathbf{H}, \mathbf{T} \rangle$. An interpretation $\mathbf{M} = \langle \mathbf{H}, \mathbf{T} \rangle$ is said to be *total* when $\mathbf{H} = \mathbf{T}$.

Given an interpretation $\mathbf{M}$ and an integer number $k > 0$, by $(\mathbf{M}, k)$ we denote a new interpretation that results from "shifting" $\mathbf{M}$ in k positions, that is, the sequence of pairs $\langle H_k, T_k \rangle, \langle H_{k+1}, T_{k+1} \rangle, \langle H_{k+2}, T_{k+2} \rangle, \dots$ Note that $(\mathbf{M}, 0) = \mathbf{M}$.

Definition 1 (satisfaction). *An interpretation* $\mathbf{M} = \langle \mathbf{H}, \mathbf{T} \rangle$ *satisfies a formula* φ, *written* $\mathbf{M} \models \varphi$, *when:*

1. $\mathbf{M} \models p$ *if* $p \in H_0$, *for any atom* p.
2. $\mathbf{M} \models \varphi \wedge \psi$ *if* $\mathbf{M} \models \varphi$ *and* $\mathbf{M} \models \psi$.
3. $\mathbf{M} \models \varphi \vee \psi$ *if* $\mathbf{M} \models \varphi$ *or* $\mathbf{M} \models \psi$.
4. $\langle \mathbf{H}, \mathbf{T} \rangle \models \varphi \to \psi$ *if* $\langle x, \mathbf{T} \rangle \not\models \varphi$ *or* $\langle x, \mathbf{T} \rangle \models \psi$ *for all* $x \in \{\mathbf{H}, \mathbf{T}\}$.
5. $\mathbf{M} \models \bigcirc\varphi$ *if* $(\mathbf{M}, 1) \models \varphi$.
6. $\mathbf{M} \models \Box\varphi$ *if* $\forall j \geq 0$, $(\mathbf{M}, j) \models \varphi$
7. $\mathbf{M} \models \Diamond\varphi$ *if* $\exists j \geq 0$, $(\mathbf{M}, j) \models \varphi$

A formula φ is *valid* if $\mathbf{M} \models \varphi$ for any $\mathbf{M}$. An interpretation $\mathbf{M}$ is a *model* of a theory Γ, written $\mathbf{M} \models \Gamma$, if $\mathbf{M} \models \alpha$, for all formula $\alpha \in \Gamma$.

We will make use of the following THT-valid equivalences:

$$\neg(F \wedge G) \leftrightarrow \neg F \vee \neg G \tag{11}$$
$$\neg(F \vee G) \leftrightarrow \neg F \wedge \neg G \tag{12}$$
$$\bigcirc(F \oplus G) \leftrightarrow \bigcirc F \oplus \bigcirc G \tag{13}$$
$$\bigcirc \otimes F \leftrightarrow \otimes \bigcirc F \tag{14}$$

for any binary connective $\oplus$ and any unary connective $\otimes$. This means that De Morgan laws (11),(12) are valid, and that we can always shift the $\bigcirc$ operator to all the operands of any connective.

[1] As shown in [9], the LTL binary operators $\mathcal{U}$ ("until") and $\mathcal{R}$ ("release") can be removed by introducing auxiliary atoms.

The logic of THT is an orthogonal combination of the logic of *Here-and-There* (HT) [13] and the (standard) linear temporal logic (LTL) [7]. On the one hand, HT is obtained by disregarding temporal operators, so that only the pair of sets of atoms $\langle H_0, T_0 \rangle$ is actually relevant and we use conditions 1-3 in Definition 1 for satisfaction of propositional theories. On the other hand, if we restrict the semantics to total interpretations, $\langle \mathbf{T}, \mathbf{T} \rangle \models \varphi$ corresponds to satisfaction of formulas $\mathbf{T} \models \varphi$ in LTL. This last correspondence allows rephrasing item 4 of Definition 1 as:

4'. $\langle \mathbf{H}, \mathbf{T} \rangle \models \varphi \rightarrow \psi$ if both (1) $\langle \mathbf{H}, \mathbf{T} \rangle \models \varphi$ implies $\langle \mathbf{H}, \mathbf{T} \rangle \models \psi$; and (2) $\mathbf{T} \models \varphi \rightarrow \psi$ in LTL.

Similarly $\langle \mathbf{H}, \mathbf{T} \rangle \models \varphi \leftrightarrow \psi$ if both (1) $\langle \mathbf{H}, \mathbf{T} \rangle \models \varphi$ iff $\langle \mathbf{H}, \mathbf{T} \rangle \models \psi$; and (2) $\mathbf{T} \models \varphi \leftrightarrow \psi$ in LTL. The following proposition can also be easily checked.

Proposition 1. *For any Γ and any $\mathbf{M} = \langle \mathbf{H}, \mathbf{T} \rangle$, if $\mathbf{M} \models \Gamma$ then $\mathbf{T} \models \Gamma$.* ⊠

We proceed now to define an ordering relation among THT models of a temporal theory, so that only the minimal ones will be *selected*. Given two interpretations $\mathbf{M} = \langle \mathbf{H}, \mathbf{T} \rangle$ and $\mathbf{M}' = \langle \mathbf{H}', \mathbf{T}' \rangle$ we say that $\mathbf{M}'$ is *lower or equal than* $\mathbf{M}$, written $\mathbf{M}' \leq \mathbf{M}$, when $\mathbf{T}' = \mathbf{T}$ and for all $i \geq 0$, $H'_i \subseteq H_i$. As usual, $\mathbf{M}' < \mathbf{M}$ stands for $\mathbf{M}' \leq \mathbf{M}$ but $\mathbf{M}' \neq \mathbf{M}$.

Definition 2 (Temporal Equilibrium Model). *An interpretation* $\mathbf{M}$ *is a* temporal equilibrium model *of a theory Γ if* $\mathbf{M}$ *is a total model of Γ and there is no other* $\mathbf{M}' < \mathbf{M}$, $\mathbf{M}' \models \Gamma$. ⊠

Note that any temporal equilibrium model is total, that is, it has the form $\langle \mathbf{T}, \mathbf{T} \rangle$ and so can be actually seen as an interpretation $\mathbf{T}$ in the standard LTL. *Temporal Equilibrium Logic* (TEL) is the logic induced by temporal equilibrium models.

When we restrict the syntax to ASP programs and HT interpretations $\langle H_0, T_0 \rangle$ we talk about (non-temporal) equilibrium models, which coincide with stable models in their most general definition [14].

3 Temporal Logic Programs

As we said in the introduction, in [9] it was shown that, by introducing auxiliary atoms, any temporal theory could be reduced to a normal form we proceed to describe. Given a signature *At*, we define a *temporal literal* as any expression in the set $\{p, \neg p, \bigcirc p, \neg \bigcirc p \mid p \in At\}$.

Definition 3 (Temporal rule). *A* temporal rule *is either:*

1. *an* initial rule *of the form*

$$B_1 \wedge \cdots \wedge B_n \rightarrow C_1 \vee \cdots \vee C_m \tag{15}$$

where all the B_i and C_j are temporal literals, $n \geq 0$ and $m \geq 0$.

2. a dynamic rule *of the form* $\Box r$*, where* r *is an initial rule.*
3. a fulfillment rule *like* $\Box(\Box p \to q)$ *or like* $\Box(p \to \Diamond q)$ *with* p, q *atoms.* ⊠

In the three cases, we respectively call rule *body* and rule *head* to the antecedent and consequent of the (unique) rule implication. In initial (resp. dynamic) rules, we may have an empty head $m = 0$ corresponding to $\bot$ – if so, we talk about an *initial* (resp. *dynamic*) *constraint.* A *temporal logic program*[2] (TLP for short) is a finite set of temporal rules. A TLP without temporal operators, that is, a set of initial rules without $\bigcirc$, is said to be an *ASP program*[3].

As an example of TLP take the program Π_1 consisting of:

$$\neg a \wedge \bigcirc b \to \bigcirc a \tag{16}$$
$$\Box(a \to b) \tag{17}$$
$$\Box(\neg b \to \bigcirc a) \tag{18}$$

where (16) is an initial rule and (17),(18) are dynamic rules.

Looking at the semantics of $\Box$ it seems clear that we can understand a dynamic rule $\Box r$ as an infinite sequence of expressions like $\bigcirc^i r$, one for each $i \geq 0$. Using (13),(14) we can shift $\bigcirc^i$ inside all connectives in r so that $\bigcirc^i r$ is equivalent to an initial rule resulting from prefixing any atom in r with $\bigcirc^i$. To put an example, if $r =$ (18) then $\bigcirc^2 r$ would correspond to $(\neg \bigcirc^2 b \to \bigcirc^3 a)$.

Definition 4 (i-expansion of a rule). *Given* $i \geq 0$*, the* i-expansion *of a dynamic rule* $\Box r$*, written* $(\Box r)^i$*, is a set of rules defined as:*

$$(\Box r)^i \overset{\text{def}}{=} \begin{cases} \emptyset & \textit{if } i = 0 \textit{ and } r \textit{ contains some `}\bigcirc\textit{'} \\ \{\bigcirc^j r \mid 0 \leq j \leq i-1\} & \textit{if } i > 0 \textit{ and } r \textit{ contains some `}\bigcirc\textit{'} \\ \{\bigcirc^j r \mid 0 \leq j \leq i\} & \textit{otherwise} \end{cases}$$

If r *is an initial rule, its* i*-expansion is defined as:*

$$r^i \overset{\text{def}}{=} \begin{cases} \emptyset & \textit{if } i = 0 \textit{ and } r \textit{ contains some `}\bigcirc\textit{'} \\ r & \textit{otherwise} \end{cases}$$
⊠

In this way, the superindex i refers to the longest sequence of $\bigcirc$'s used in the rule. For instance, $(18)^3$ would be:

$$\{\ \ (\neg b \to \bigcirc a),\ \ (\neg \bigcirc b \to \bigcirc^2 a),\ \ (\neg \bigcirc^2 b \to \bigcirc^3 a)\ \ \}$$

We extend this notation to programs, so that given a TLP Π its i-expansion Π^i results from replacing each initial or dynamic rule r in Π by r^i. An interesting observation is that we can understand each Π^i as a (non-temporal) ASP program for signature $At^i \overset{\text{def}}{=} \{\text{“}\bigcirc^j p\text{”} \mid p \in At, 0 \leq j \leq i\}$ where we understand each “$\bigcirc^j p$” as a different propositional atom. This same notation can be applied

[2] In fact, as shown in [9], this normal form can be even more restrictive: initial rules can be replaced by atoms, and we can avoid the use of literals of the form $\neg \bigcirc p$.
[3] In ASP literature, this is called a a disjunctive program with negation in the head.

to interpretations. If $\mathbf{T}$ is an LTL interpretation (an infinite sequence of sets of atoms) for signature At its i-expansion would be the corresponding propositional interpretation for signature At^i defined as $\mathbf{T}^i \stackrel{\text{def}}{=} \{\bigcirc^j p \mid 0 \leq j \leq i, p \in T_j\}$ and if $\mathbf{M} = \langle \mathbf{H}, \mathbf{T} \rangle$ is a THT interpretation then its i-expansion is defined as the HT interpretation $\mathbf{M}^i \stackrel{\text{def}}{=} \langle \mathbf{H}^i, \mathbf{T}^i \rangle$. In all these cases, we also define the ω-*expansion* (or simply, *expansion*) as the infinite union of all i-expansions for all $i \geq 0$. Thus, for instance $(\Box r)^\omega \stackrel{\text{def}}{=} \bigcup_{i \geq 0} (\Box r)^i$ and similarly for Π^ω, At^ω, $\mathbf{T}^\omega$ and $\mathbf{M}^\omega$. It is interesting to note that, for any classical interpretation $\mathbf{T}'$ for signature At^ω, we can always build a corresponding LTL interpretation $\mathbf{T}$ in signature At such that $\mathbf{T}^\omega = \mathbf{T}'$. The following theorem establishes the correspondence between a temporal program and its expansion.

Theorem 1. *Let Π be a TLP without fulfillment rules. Then $\langle \mathbf{T}, \mathbf{T} \rangle$ is a temporal equilibrium model of Π under signature At iff $\mathbf{T}^\omega$ is a stable model of Π^ω under signature At^ω.* $\boxtimes$

The above theorem allows us reading a TLP with initial and dynamic rules as an ASP program with infinite "copies" of the same rule schemata. In many cases, this allows us to foresee the temporal equilibrium models of a TLP. For instance, if we look at our example TLP Π_1, it is easy to see that we should get $T_0 = \emptyset$ as the only rule affecting the situation $i = 0$ is $(17)^0 = (a \rightarrow b)$. For situation $i = 1$ we would have rules $(\bigcirc a \rightarrow \bigcirc b) \in (17)^1$ and $(\neg b \rightarrow \bigcirc a) \in (18)^1$ so that, given T_0 we obtain $\bigcirc a \wedge \bigcirc b$, that is, $T_1 = \{a, b\}$. For $i = 2$, the involved rules are $(\bigcirc^2 a \rightarrow \bigcirc^2 b) \in (17)^2$ and $(\neg \bigcirc b \rightarrow \bigcirc^2 a) \in (18)^2$ so that, given T_1 we obtain $T_2 = \emptyset$. In a similar way, for $i = 3$ we have rules $(\bigcirc^3 a \rightarrow \bigcirc^3 b)$ and $(\neg \bigcirc^2 b \rightarrow \bigcirc^3 a)$ leading to $T_3 = \{a, b\}$ and then this behaviour is repeated. To sum up, we get a unique temporal equilibrium model $\langle \mathbf{T}, \mathbf{T} \rangle$ for Π_1 where $\mathbf{T}$ can be captured by the regular expression $(\ \emptyset\ \{a, b\}\ )^+$.

In some cases, however, we may face new situations that are not common in standard ASP. For instance, consider the TLP Π_2 consisting of the two rules (9), (10). This program has no temporal equilibrium models. To see why, note that Π_2 is equivalent to $\Box(\neg \bigcirc p \vee \bigcirc p \rightarrow p)$ that, in its turn, is LTL-equivalent to $\Box p$. Thus, the only LTL-model $\mathbf{T}$ of Π_2 has the form $T_i = \{p\}$ for any $i \geq 0$. However, it is easy to see that the interpretation $\langle \mathbf{H}, \mathbf{T} \rangle$ with $H_i = \emptyset$ for all $i \geq 0$ is also a THT model, whereas $\mathbf{H} < \mathbf{T}$. Note that, by Theorem 1, this means that the ASP program Π_2^ω has no stable models, although it is an *acyclic program* and (finite) acyclic programs always have a stable model. The intuitive reason for this is that atoms $\bigcirc^i p$ infinitely depend on the future, and there is no way to build a founded reasoning starting from facts or the absence of them at a given end point[4].

[4] In fact, this example was extracted from a first-order counterpart, the pair of rules $\neg p(s(X)) \rightarrow p(X)$ and $p(s(X)) \rightarrow p(X)$, that were used in [15] to show that an acyclic program without a well-founded dependence ordering relation may have no stable models.

4 Splitting a Temporal Logic Program

Fortunately, most ASP programs dealing with transition systems represent rules so that past does not depend on the future. This is what we called *future projected dependence* and can be captured by the following subclass of TLPs.

Definition 5 (Splitable TLP). *A TLP Π for signature At is said to be* splitable *if Π consists of rules of any of the forms:*

$$B \wedge N \rightarrow H \tag{19}$$

$$B \wedge \bigcirc B' \wedge N \wedge \bigcirc N' \rightarrow \bigcirc H' \tag{20}$$

$$\Box(B \wedge \bigcirc B' \wedge N \wedge \bigcirc N' \rightarrow \bigcirc H') \tag{21}$$

where B and B' are conjunctions of atoms, N and N' are conjunctions of negative literals like $\neg p$ with $p \in At$, and H and H' are disjunctions of atoms. ⊠

The set of rules of form (19) in Π will be denoted $ini_0(\Pi)$ and correspond to initial rules for situation 0. The rules of form (20) in Π will be represented as $ini_1(\Pi)$ and are initial rules for the transition between situations 0 and 1. Finally, the set of rules of form (21) is written $dyn(\Pi)$ and contains dynamic rules. Both in (20) and (21), we understand that operator $\bigcirc$ is actually shifted until it only affects to atoms – this is always possible due to equivalences (13), (14). We will also use the formulas B, B', N, N', H and H' as sets, denoting the atoms that occur in each respective formula.

Notice that a rule of the form $\Box(B \wedge N \rightarrow H)$ (i.e., without $\bigcirc$ operator) is not splitable but can be transformed into the equivalent pair of rules $B \wedge N \rightarrow H$ and $\Box(\bigcirc B \wedge \bigcirc N \rightarrow \bigcirc H)$ which are both splitable. For instance, (17) becomes the pair of rules:

$$a \rightarrow b \tag{22}$$

$$\Box(\bigcirc a \rightarrow \bigcirc b) \tag{23}$$

As an example, Π_2=(9)-(10) is not splitable, whereas Π_1=(16),(18),(22),(23) is splitable being $ini_0(\Pi_1)$=(22), $ini_1(\Pi_1)$=(16) and $dyn(\Pi_1)$=(22),(18). In particular, in (16) we have the non-empty sets $B' = \{b\}$, $N = \{a\}$ and $H' = \{a\}$, whereas for (18) the sets are $N = \{b\}$, $H' = \{a\}$. It can be easily seen that the rules (5)-(8) are also in splitable form.

As we explained in the introduction, the most interesting feature of splitable TLPs is that we can apply the so-called *splitting* technique [10] to obtain their temporal equilibrium models in an incremental way. Let us briefly recall this technique for the case of ASP programs. Following [10] we define:

Definition 6 (Splitting set). *Let Π be an ASP program consisting of (non-temporal) rules like (19). Then a set of atoms U is a* splitting set *for Π if, for any rule like (19) in Π: if $H \cap U \neq \emptyset$ then $(B \cup N \cup H) \subseteq U$. The set of rules satisfying $(B \cup N \cup H) \subseteq U$ are denoted as $b_U(\Pi)$ and called the* bottom *of Π with respect to U.* ⊠

Consider the program:

$$a \rightarrow c \tag{24}$$
$$b \rightarrow d \tag{25}$$
$$\neg b \rightarrow a \tag{26}$$
$$\neg a \rightarrow b \tag{27}$$

The set $U = \{a, b\}$ is a splitting set for Π being $b_U(\Pi) = \{(26), (27)\}$. The idea of splitting is that we can compute first each stable model X of $b_U(\Pi)$ and then use the truth values in X for simplifying the program $\Pi \setminus b_U(\Pi)$ from which the rest of truth values for atoms not in U can be obtained. Formally, given $X \subseteq U \subseteq At$ and an ASP program Π, for each rule r like (19) in Π such that $B \cap U \subseteq X$ and $N \cap U$ is disjoint from X, take the rule $r^\bullet : B^\bullet \wedge N^\bullet \rightarrow H$ where $B^\bullet = (B \setminus U)$ and $N^\bullet = (N \setminus U)$. The program consisting of all rules $r^\bullet$ obtained in this way is denoted as $e_U(\Pi, X)$. Note that this program is equivalent to replacing in all rules in Π each atom $p \in U$ by $\bot$ if $p \notin X$ and by $\top$ if $p \in X$.

In the previous example, the stable models of $b_U(\Pi)$ are $\{a\}$ and $\{b\}$. For the first stable model $X = \{a\}$, we get $e_U(\Pi \setminus b_U(\Pi), \{a\}) = \{\top \rightarrow c\}$ so that $X \cup \{c\} = \{a, c\}$ should be a stable model for the complete program Π. Similarly, for $X = \{b\}$ we get $e_U(\Pi \setminus b_U(\Pi), \{b\}) = \{\top \rightarrow d\}$ and a "completed" stable model $X \cup \{d\} = \{b, d\}$. The following result guarantees the correctness of this method in the general case.

Theorem 2 (from [10]). *Let U be a splitting set for a set of rules Π like (19). A set of atoms X is an stable model of Π if, and only if both*

(i) *$X \cap U$ is a stable model of $b_U(\Pi)$;*
(ii) *and $X \setminus U$ is a stable model of $e_U(\Pi \setminus b_U(\Pi), X \cap U)$.* $\boxtimes$

In [10] this result was generalised for an infinite sequence of splitting sets, showing an example of a logic program with variables and a function symbol, so that the ground program was infinite. We adapt next this splitting sequence result for the case of splitable TLPs in TEL.

From Definition 4 we can easily conclude that, when Π is a splitable TLP, its programs expansions have the form $\Pi^0 = ini_0(\Pi)$ and $\Pi^i = ini_0(\Pi) \cup ini_1(\Pi) \cup dyn(\Pi)^i$ for $i > 0$.

Proposition 2. *Given a splitable TLP Π for signature At and any $i \geq 0$:*

(i) *At^i is a splitting set for Π^ω;*
(ii) *and $b_{At^i}(\Pi^\omega) = \Pi^i$.* $\boxtimes$

Given any rule like r like (20) of (21) and a set of atoms X, we define its *simplification* $simp(r, X)$ as:

$$simp(r, X) \stackrel{\text{def}}{=} \begin{cases} \bigcirc B' \wedge \bigcirc N' \rightarrow \bigcirc H' & \text{if } B \subseteq X \text{ and } N \cap X = \emptyset \\ \top & \text{otherwise} \end{cases}$$

Given some LTL interpretation $\mathbf{T}$, let us define now the sequence of programs:

$$\Pi[\mathbf{T}, i] \stackrel{\text{def}}{=} e_{At^i}\left(\Pi^\omega \setminus \Pi^i\,,\, \mathbf{T}^i\right)$$

that is, $\Pi[\mathbf{T}, i]$ is the "simplification" of Π^ω by replacing atoms in At^i by their truth value with respect to $\mathbf{T}^i$. Then, we have:

Proposition 3

$$\begin{aligned}\Pi[\mathbf{T}, 0] &= (dyn(\Pi)^\omega \setminus dyn(\Pi)^1) \cup \{simp(r, T_0) \mid r \in ini_1(\Pi) \cup dyn(\Pi)\}\\ \Pi[\mathbf{T}, i] &= (dyn(\Pi)^\omega \setminus dyn(\Pi)^{i+1}) \cup \ \{\bigcirc^i simp(r, T_i) \mid r \in dyn(\Pi)\}\end{aligned}$$

for any $i \geq 1$. ⊠

As we can see, programs $\Pi[\mathbf{T}, i]$ maintain most part of $dyn(\Pi)^\omega$ and only differ in simplified rules. Let us call these sets of simplified rules:

$$\begin{aligned}slice(\Pi, \mathbf{T}, 0) &\stackrel{\text{def}}{=} \Pi^0 = ini_0(\Pi)\\ slice(\Pi, \mathbf{T}, 1) &\stackrel{\text{def}}{=} \{simp(r, T_0) \mid r \in ini_1(\Pi) \cup dyn(\Pi)\}\\ slice(\Pi, \mathbf{T}, i+1) &\stackrel{\text{def}}{=} \{\bigcirc^i simp(r, T_i) \mid r \in dyn(\Pi)\} \qquad \text{for } i \geq 1\end{aligned}$$

Theorem 3 (Splitting Sequence Theorem). *Let $\langle \mathbf{T}, \mathbf{T}\rangle$ be a model of a splitable TLP Π. $\langle \mathbf{T}, \mathbf{T}\rangle$ is a temporal equilibrium model of Π iff*

(i) $\mathbf{T}^0 = T_0$ *is a stable model of* $slice(\Pi, \mathbf{T}, 0) = \Pi^0 = ini_0(\Pi)$ *and*
(ii) $(\mathbf{T}^1 \setminus At^0)$ *is a stable model of* $slice(\Pi, \mathbf{T}, 1)$ *and*
(iii) $(\mathbf{T}^i \setminus At^{i-1})$ *is a stable model of* $slice(\Pi, \mathbf{T}, i)$ *for* $i \geq 2$. ⊠

As an example, let us take again program Π_1 = (16), (22), (23), (18). The program $\Pi_1^0 = ini_0(\Pi_1)$ = (22) has the stable model $\mathbf{T}^0 = \emptyset = T_0$. Then we take $slice(\Pi, \mathbf{T}, 1) = \{simp(r, T_0) \mid r \in ini_1(\Pi) \cup dyn(\Pi)\}$ that corresponds to $\{(\bigcirc b \rightarrow \bigcirc a),\ (\bigcirc a \rightarrow \bigcirc b),\ (\top \rightarrow \bigcirc a)\}$ whose stable model is $\{\bigcirc a, \bigcirc b\} = (\mathbf{T}^1 \setminus At^0)$ so that $T_1 = \{a, b\}$. In the next step, $slice(\Pi, \mathbf{T}, 2) = \{\bigcirc simp(r, T_1) \mid r \in dyn(\Pi)\} = \{(\bigcirc^2 a \rightarrow \bigcirc^2 b), (\top)\}$ whose stable model is $\emptyset = (\mathbf{T}^2 \setminus At^1)$ so that $T_2 = \emptyset$. Then, we would go on with $slice(\Pi, \mathbf{T}, 3) = \{\bigcirc^2 simp(r, T_2) \mid r \in dyn(\Pi)\} = \{(\bigcirc^3 a \rightarrow \bigcirc^3 b), (\top \rightarrow \bigcirc^3 a)\}$ leading to $\{\bigcirc^3 a, \bigcirc^3 b\}$ that is $T_3 = \{a, b\}$ and so on.

5 Loop Formulas

Theorem 3 allows us building the temporal equilibrium models by considering an infinite sequence of finite ASP programs $slice(\Pi, \mathbf{T}, i)$. If we consider each program $\Pi' = slice(\Pi, \mathbf{T}, i+1)$ for signature $At^{i+1} \setminus At^i$ then, since it is a standard disjunctive ASP program, we can use the main result in [12] to compute its stable models by obtaining the classical models of a theory $\Pi' \cup LF(\Pi')$ where LF stands for *loop formulas*. To make the paper self-contained, we recall next some definitions and results from [12].

Given an ASP program Π we define its *(positive) dependency graph* $G(\Pi)$ where its vertices are At (the atoms in Π) and its edges are $E \subseteq At \times At$ so that $(p, p) \in E$ for any atom[5] p, and $(p, q) \in E$ if there is an ASP rule in Π like (19) with $p \in H$ and $q \in B$. A nonempty set L of atoms is called a *loop* of a program Π if, for every pair p, q of atoms in L, there exists a path from p to q in $G(\Pi)$ such that all vertices in the path belong to L. In other words, L is a loop of iff the subgraph of $G(\Pi)$ induced by L is strongly connected. Notice that reflexivity of $G(\Pi)$ implies that for any atom p, the singleton $\{p\}$ is also a loop.

Definition 7 (external support). *Given an ASP program Π for signature At, the* external support *formula of a set of atoms $Y \subseteq At$ with respect to Π, written $ES_\Pi(Y)$ is defined by:*

$$\bigvee_{r \in R(Y)} \left(B \wedge N \wedge \bigwedge_{p \in H \setminus Y} \neg p \right)$$

where $R(Y) = \{r \in \Pi$ like (19) $\mid H \cap Y \neq \emptyset$ and $B \cap Y = \emptyset\}$. ⊠

Theorem 4 (from [12]). *Given a program Π for signature At, and a (classical) model $X \subseteq At$ of Π then X is a stable model of Π iff for every loop Y of Π, X satisfies $\bigvee_{p \in Y} p \rightarrow ES_\Pi(Y)$* ⊠

This result can be directly applied to each finite ASP program $slice(\Pi, \mathbf{T}, i)$. As we have slice programs for $i = 0, 1, \ldots$, this means we would obtain an infinite sequence of classical theories (each program plus its loop formulas). Fortunately, these theories are not arbitrary. For situations $0, 1$, we may obtain loops induced by dependencies that are due to initial rules, but for $i \geq 2$ loops follow a *repetitive pattern*, so they can be easily captured using $\Box$ and $\bigcirc$ operators. Thus, we just need to consider loops for situations $0, 1, 2$ bearing in mind that any loop at level $i = 2$ will occur repeatedly from then on. Given a splitable TLP Π its associated dependency graph $G(\Pi)$ is generated from the expanded (ASP) program Π^2, so that its nodes are atoms in the signature At^2 and its loops are obtained from this finite program. For instance, given our example program Π_1, its graph $G(\Pi_1)$, shown in Figure 1, is obtained from the expanded program Π_1^2 and, as we can see, contains the loops $\{\bigcirc a, \bigcirc b\}$ plus $\{A\}$ for any $A \in At^2$.

Theorem 5. *Let Π be a splitable TLP and $\mathbf{T}$ an LTL model of Π. Then $\langle \mathbf{T}, \mathbf{T} \rangle$ is a temporal equilibrium model of Π iff $\mathbf{T}$ is an LTL model of the union of formulas $LF(Y)$ defined as:*

$$\begin{array}{ll} Y^\vee \rightarrow ES_{ini_0(\Pi)}(Y) & \textit{for any loop } Y \subseteq At^0 = At \\ Y^\vee \rightarrow ES_{ini_1(\Pi) \cup dyn(\Pi)^1}(Y) & \textit{for any loop } Y \subseteq (At^1 \setminus At^0) \\ \Box\Big(Y^\vee \rightarrow ES_{dyn(\Pi)^2 \setminus dyn(\Pi)^1}(Y)\Big) & \textit{for any loop } Y \subseteq (At^2 \setminus At^1) \end{array}$$

⊠

[5] The original formulation in [12] did not consider reflexive edges, dealing instead with the idea of paths of length 0.

$$\begin{tikzcd}
a & \bigcirc a \arrow[d, dashed, bend right] & \bigcirc^2 a \\
b \arrow[u] & \bigcirc b \arrow[u, dashed, bend right] & \bigcirc^2 b \arrow[u]
\end{tikzcd}$$

Fig. 1. Graph $G(\Pi_1)$ (reflexive arcs are not displayed) corresponding to Π_1^2

Proof. For a proof sketch, notice that each loop is always inside some $slice(\Pi, \mathbf{T}, i)$; otherwise, some past situation would necessarily depend on the future. In this way, all atoms in At^{i-1} are external to the loop. This means, for instance, that Y is a loop of $ini_1(\Pi) \cup dyn(\Pi)^1$ iff Y is a loop of $slice(\Pi, \mathbf{T}, 1)$. Also, note that if we have some rule as (20) in $ini_1(\Pi)$ or as (21) in $dyn(\Pi)$ and $\langle \mathbf{T}, \mathbf{T} \rangle \models \Pi$, then the assertion $T_0 \models B \wedge \bigcirc B' \wedge N \wedge \bigcirc N' \wedge \neg (\bigcirc H' \setminus Y)^\vee$ is equivalent to requiring both $(T^1 \setminus At^0) \models \bigcirc B' \wedge \bigcirc N' \wedge \neg (\bigcirc H' \setminus Y)^\vee$ and $\bigcirc B' \wedge \bigcirc N' \rightarrow \bigcirc H' \in slice(\Pi, \mathbf{T}, 1)$. As a consequence $\langle \mathbf{T}, \mathbf{T} \rangle \models Y^\vee \rightarrow ES_{ini_1(\Pi) \cup dyn(\Pi)^1}(Y)$ iff $(\mathbf{T}^1 \setminus At^0) \models Y^\vee \rightarrow ES_{slice(\Pi, \mathbf{T}, 1)}(Y)$ for any loop $Y \subseteq (At^1 \setminus At^0)$. Since the first two slices are affected by initial rules, they are specified in a separate case, whereas from slice 2 on we get a repetitive pattern using $\Box$. $\boxtimes$

In our running example Π_1 we have $At^0 = \{a, b\}$ and $ini_0(\Pi) =$ (22) with two loops $\{a\}, \{b\}$ where $LF(\{a\}) = (a \rightarrow \bot)$ and $LF(\{b\}) = (b \rightarrow a)$. For $(At^1 \setminus At^0) = \{\bigcirc a, \bigcirc b\}$ we take the program $ini_1(\Pi_1) \cup dyn(\Pi)^1)$, that is, rules (16), (23), (18) ignoring $\Box$. We get three loops leading to the corresponding loop formulas $(\bigcirc a \rightarrow (\neg a \wedge \bigcirc b) \vee \neg b)$, $(\bigcirc b \rightarrow \bigcirc a)$ and $(\bigcirc a \vee \bigcirc b \rightarrow \neg b)$. Finally, for $At^2 \setminus At^1$ we have two loop formulas $\Box(\bigcirc^2 b \rightarrow \bigcirc^2 a)$ and $\Box(\bigcirc^2 a \rightarrow \neg \bigcirc b)$. It is not difficult to see that $\Pi_1 \cup LF(\Pi_1)$ is equivalent to the LTL theory: $\neg a \wedge \neg b \wedge \Box(\bigcirc a \leftrightarrow \neg b) \wedge \Box(\bigcirc b \leftrightarrow \neg b)$.

6 Conclusions

We have presented a class of temporal logic programs (that is ASP programs with temporal operators) for which their temporal equilibrium models (the analogous to stable models) can be computed by translation to LTL. To this aim, we have combined two well-known techniques in the ASP literature called splitting and loop formulas. This syntactic class has as restriction so that rule heads never refer to a temporally previous situation than those referred in the body. Still, it is expressive enough to cover most ASP examples dealing with dynamic scenarios.

We have implemented a system, called `STeLP` that uses this technique to translate a program and calls an LTL model checker to obtain its temporal equilibrium models, in the form of a Büchi automaton. This tool allows some practical features like dealing with variables or specifying fluents and actions. More examples and information can be obtained in [16].

References

1. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
2. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm, pp. 169–181. Springer, Heidelberg (1999)
3. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) Logic Programming: Proc. of the Fifth International Conference and Symposium, vol. 2, pp. 1070–1080. MIT Press, Cambridge (1988)
4. Cabalar, P., Pérez Vega, G.: Temporal equilibrium logic: A first approach. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 241–248. Springer, Heidelberg (2007)
5. Pearce, D.: A new logical characterisation of stable models and answer sets. In: Dix, J., Przymusinski, T.C., Moniz Pereira, L. (eds.) NMELP 1996. LNCS (LNAI), vol. 1216. Springer, Heidelberg (1997)
6. Pearce, D.: Equilibrium logic. Annals of Mathematics and Artificial Intelligence 47(1-2), 3–41 (2006)
7. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1991)
8. Aguado, F., Cabalar, P., Pérez, G., Vidal, C.: Strongly equivalent temporal logic programs. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 8–20. Springer, Heidelberg (2008)
9. Cabalar, P.: A normal form for linear temporal equilibrium logic. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 64–76. Springer, Heidelberg (2010)
10. Lifschitz, V., Turner, H.: Splitting a logic program. In: Proceedings of the 11th International Conference on Logic programming (ICLP 1994), pp. 23–37 (1994)
11. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. In: Artificial Intelligence, pp. 112–117 (2002)
12. Ferraris, P., Lee, J., Lifschitz, V.: A generalization of the Lin-Zhao theorem. Annals of Mathematics and Artificial Intelligence 47, 79–101 (2006)
13. Heyting, A.: Die formalen Regeln der intuitionistischen Logik. In: Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse, pp. 42–56 (1930)
14. Ferraris, P.: Answer sets for propositional theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 119–131. Springer, Heidelberg (2005)
15. Fages, F.: Consistency of Clark's completion and existence of stable models. Methods of Logic in Computer Science 1, 51–60 (1994)
16. Cabalar, P., Diéguez, M.: STeLP – a tool for temporal answer set programming. In: Delgrande, J., Faber, W. (eds.) LPNMR 2011. LNCS (LNAI), vol. 6645, pp. 359–364. Springer, Heidelberg (2011)

Pushing Efficient Evaluation of HEX Programs by Modular Decomposition[*]

Thomas Eiter[1], Michael Fink[1], Giovambattista Ianni[2], Thomas Krennwallner[1], and Peter Schüller[1]

[1] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink,tkren,ps}@kr.tuwien.ac.at

[2] Dipartimento di Matematica, Cubo 30B,
Università della Calabria 87036 Rende (CS), Italy
ianni@mat.unical.it

Abstract. The evaluation of logic programs with access to external knowledge sources requires to interleave external computation and model building. Deciding where and how to stop with one task and proceed with the next is a difficult problem, and existing approaches have severe scalability limitations in many real-world application scenarios. We introduce a new approach for organizing the evaluation of logic programs with external knowledge sources and describe a configurable framework for dividing the non-ground program into overlapping possiblysmaller parts called evaluation units. These units will then be processed by interleaving external evaluations and model building according to an evaluation and a model graph, and by combining intermediate results. Experiments with our prototype implementation show a significant improvement of this technique compared to existing approaches. Interestingly, even for ordinary logic programs (with no external access), our decomposition approach speeds up existing state of the art ASP solvers in some cases, showing its potential for wider usage.

1 Introduction

Motivated by a need for knowledge bases to access external sources, extensions of declarative KR formalisms have been conceived that provide this capability, which is often realized via an API like interface. In particular, HEX programs [6] extend non-monotonic logic programs under the stable model semantics, with the possibility to bidirectionally access external sources of knowledge and/or computation. E.g., a rule

$$\mathit{pointsTo}(X,Y) \leftarrow \&\mathit{hasHyperlink}[X](Y), \mathit{url}(X)$$

might be used for obtaining pairs of URLs (X,Y), where X actually links Y on the Web, and $\&\mathit{hasHyperlink}$ is an *external predicate* construct. Besides constant values, as above, also relational knowledge (predicate extensions) can flow from external sources to the logic program at hand and vice versa, and recursion involving external predicates is allowed under suitable safety conditions. This facilitates a variety of applications which require logic programs to interact with external environments, such as

[*] This research has been supported by the Austrian Science Fund (FWF) project P20841 and the Vienna Science and Technology Fund (WWTF) project ICT08-020.

querying RDF sources using SPARQL [17], bioinformatics [11], combining rules and ontologies [5], e-government [21], planning [14], and multi-contextual reasoning [2], to mention a few.

Despite the lack of function symbols, an unrestricted use of external atoms leads to undecidability, as new constants may be introduced, yielding a potentially infinite Herbrand universe. However, even under suitable restrictions like domain-expansion safety [7], the efficient evaluation of HEX-programs is challenging, due to aspects like nonmonotonic atoms and recursive access (e.g., in transitive closure computations).

Advanced in this regard was [7], which fostered an evaluation approach using a traditional LP system. Roughly, the values of ground external atoms are guessed, model candidates are computed as answer sets of a rewritten program, and then those discarded which violate the guess. A generalized notion of Splitting Set [13] was introduced in [7] for non-ground HEX-programs, which were then split into subprograms with and without external access, where the former are as large and the latter as small as possible. They are evaluated with various specific techniques, depending on their structure [7,20]. However, for real-world applications this approach has severe scalability limitations, as the number of ground external atoms may be large, and their combination causes a huge number of model candidates and memory outage without any answer set output.

To remedy this problem, we reconsider model computation and make several contributions, which are summarized as follows.

• We present an evaluation framework for HEX-programs, which allows for flexible program evaluation. It comprises an *evaluation graph*, which captures a modular decomposition and partial evaluation order, and a *model graph*, which comprises for each node, sets of input models (which need to be combined) and output models to be passed on. This structure allows us to realize customized divide-and-conquer evaluation strategies, using a further generalization of the Splitting Theorem. As the method works on nonground programs, value introduction by external calculations and applying optimization techniques based on domain splitting [4] are feasible.
• The nodes in the evaluation graph are *evaluation units* (program sub-modules), which noticeably — and different from other decomposition approaches — may overlap and be non-maximal resp. minimal. In particular, constraint sharing can prune irrelevant partial models and candidates earlier than in previous approaches.
• A prototype of the evaluation framework has been implemented, which is generic and can be instantiated with different ASP solvers (in our case, with dlv and clasp). It features also *model streaming*, i.e., computation one by one. In combination with early model pruning, this can considerably reduce memory consumption and avoid termination without solution output in a larger number of settings.
• In contrast to the previous approach, the new one allows for generating parallelizable evaluation plans. Applying it to ordinary programs (without external functions) allows us to do parallel solving with a solver software that does not have parallel computing capabilities itself ("parallelize from outside").

In order to assess the new approach, we conducted a series of experiments which clearly demonstrate its usefulness. The new implementation outperforms the current dlvhex system significantly, using (sometimes exponentially) less memory and running much faster. Interestingly, also on some ordinary test programs it compared well to state of the art ASP solvers: apart from some overhead on fast solved instances, our decomposition approach showed a speed up on top of dlv and clasp. The results indicate a potential for widening the optimization techniques of ordinary logic programs, and possibly also other formalisms like multi-context systems. Due to space limitation, only selected experiments are shown and proofs omitted. The full experimental outcome with benchmark instances is available at `http://www.kr.tuwien.ac.at/research/systems/dlvhex/experiments.html`.

2 Preliminaries

HEX programs [7] are built on mutually disjoint sets $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ of *constant*, *variable*, and *external predicate names*, respectively. By default, the elements of $\mathcal{X}$ (resp., $\mathcal{C}$) start with a letter in upper (resp., lower) case; elements of $\mathcal{G}$ are prefixed with '&'. Constant names serve both as individual and predicate names. Noticeably, $\mathcal{C}$ may be infinite. *Terms* are elements of $\mathcal{C} \cup \mathcal{X}$. A (*higher-order*) *atom* is of form $Y_0(Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms; $n \geq 0$ is its *arity*. The atom is *ordinary*, if $Y_0 \in \mathcal{C}$. In this paper, we assume that all atoms are ordinary, i.e., of the form $p(Y_1, \ldots, Y_n)$. An *external atom* is of the form $\&g[\mathbf{X}](\mathbf{Y})$, where $\mathbf{X} = X_1, \ldots, X_n$ and $\mathbf{Y} = Y_1, \ldots, Y_m$ are lists of terms (called *input* and *output list*, resp.), and $\&g$ is an *external predicate name*.

HEX-programs (or simply *programs*) are finite sets of *rules* r of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, not\ \beta_{n+1}, \ldots, not\ \beta_m, \tag{1}$$

where $m, k \geq 0$, all α_i are atoms, all β_j are atoms or external atoms, and "*not*" is *negation as failure* (or *default negation*). If $k = 0$, r is a *constraint*, otherwise a *non-constraint*. If r is variable-free, $k = 1$, and $m = 0$, it is a *fact*.

We call $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ the *head* of r and $B(r) = B^+(r) \cup B^-(r)$ the *body* of r, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$ are the (sets of) *positive* and *negative body atoms*, respectively. We write $a \sim b$ when two atoms a and b unify.

Semantics. Answer sets of ordinary programs [10] are extended to HEX-programs P, using the FLP reduct [8]. The *Herbrand base* HB_P of P, is the set of all ground instances of atoms and external atoms occurring in P, obtained by a variable substitution over $\mathcal{C}$. The grounding of a rule r, $grnd(r)$, and of P, $grnd(P) = \bigcup_{r \in P} grnd(r)$, is analogous.

An *interpretation relative to* P is any subset $I \subseteq HB_P$ containing only atoms. Satisfaction is defined as follows: I is a *model* of (i) an atom $a \in HB_P$ respectively (ii) a ground external atom $a = \&g[\mathbf{x}](\mathbf{y})$, denoted $I \models a$, iff (i) $a \in I$ respectively (ii) $f_{\&g}(I, \mathbf{x}, \mathbf{y}) = 1$, where $f_{\&g} \colon 2^{HB_P} \times \mathcal{C}^{n+m} \to \{0, 1\}$ is a (fixed) function associated with $\&g$, called *oracle function*; intuitively, $f_{\&g}$ tells whether $\mathbf{y}$ is in the output computed by the external source $\&g$ on input $\mathbf{x}$.

For a ground rule r, (i) $I \models H(r)$ iff $I \models a$ for some $a \in H(r)$, (ii) $I \models B(r)$ iff $I \models a$ for every $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ or $I \not\models B(r)$. Then, I is a *model* of P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$.

The *FLP-reduct* [8] of P w.r.t. an interpretation I, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$. Finally, $I \subseteq HB_P$ is an *answer set of* P, iff I is a subset-minimal model of fP^I. By $\mathcal{AS}(P)$ we denote the set of all answer sets of P.

A ground external atom a is called *monotonic*, iff $I \models a$ implies $I' \models a$ for all interpretations I, I' such that $I \subseteq I'$. A non-ground external atom is *monotonic*, if all its ground instances are. For practical concerns, we assume that each input argument X_i of an external atom $a = \&g[\mathbf{X}](\mathbf{Y})$, has a type label *predicate* or *constant*, which is unique for $\&g$. We then assume that $\forall I, I', \mathbf{x}, \mathbf{y}. f_{\&g}(I, \mathbf{x}, \mathbf{y}) = f_{\&g}(I', \mathbf{x}, \mathbf{y})$ holds for all I and I' which coincide on all the extensions of predicates x_i such that X_i is of type *predicate*; hence, $f_{\&g}$ depends only on the input of a given by predicate extensions and individuals.

3 Formal Framework

The former HEX evaluation algorithm [7] is based on a dependency graph between non-ground atoms; depending on the above, piecewise evaluation is carried out on appropriate selections of sets of rules (the 'bottoms' of a program). In contrast with that, we base evaluation on dependency between, possibly overlapping, subsets of rules of the program at hand. We call such groups of rules *evaluation units* (in short: units); with respect to the former approach, units are not necessarily maximal. Instead, when forming units we require that their partial models (i.e., atoms in heads of their rules) do not interfere with those of other units. This allows for independence, efficient storage, and easy composition of partial models of distinct units.

Creating an *evaluation graph* (a graph of units) for a given HEX program is done by an *evaluation heuristics*. Note that several possible evaluation graphs exist with highly varying evaluation performance. Here, we concentrate on the evaluation framework, leaving the design of optimal heuristics subject to future work. In Section 5 we informally describe the old heuristics *H1* and a simple new heuristics *H2* used in the experiments.

For illustrating our contribution, we make use of the following running example.

Example 1. Consider the HEX program P with facts $choose(a, c, d)$ and $choose(b, e, f)$:

$$\begin{array}{lrl}
r_1: & plan(a) \vee plan(b) & \leftarrow \\
r_2: & need(p,C) & \leftarrow \&cost[plan](C) \\
r_3: & use(X) \vee use(Y) & \leftarrow plan(P), choose(P,X,Y) \\
r_4: & need(u,C) & \leftarrow \&cost[use](C) \\
c_5: & & \leftarrow need(\_,money)
\end{array}$$

External atom *&cost* has one predicate type input: for any interpretation I and some constant (predicate) q, $f_{\&cost}(I, q, C) = 1$ iff $C = money$ and $I \cap \{q(a), q(f)\} \neq \emptyset$, or $C = time$ and $I \cap \{q(b), q(c), q(d), q(e)\} \neq \emptyset$; otherwise 0.

The program P informally expresses to guess a $plan$ and a sub-plan use; resource usage is evaluated using *&cost*, solutions that require $money$ are forbidden by a constraint, choosing a or f in r_1 or r_3 requires $money$, other choices require $time$. The answer set of P is $\{plan(b), use(e), need(p, time), need(u, time)\}$ (omitting facts). □

We next introduce a new notion of dependencies in HEX programs.

Dependency Information. To account for dependencies between heads and bodies of rules is a common approach for devising an operational semantics of ordinary logic programs, as done e.g. by *stratification* and its refinements like *local* [18] or *modular* *stratification* [19], or by *splitting sets* [13]. New types of dependencies were considered in [7], as in HEX programs, head-body dependencies are not the only possible source of predicate interaction. In contrast to the traditional notion of dependency that in essence hinges on propositional programs, we consider relationships between non-ground atoms. We lift the definition of atom dependency in [7] to dependencies among rules.

Definition 1 (Rule dependencies). *Let P be a program with rules $r, s \in P$. We denote by $r \rightarrow_p s$ (resp. $r \rightarrow_n s$) that r* depends positively *(resp.* negatively*)* on *s whenever:*

(i) *$a \in B^+(r)$, $b \in H(s)$, and $a \sim b$, then $r \rightarrow_p s$;*
(ii) *$a \in B^-(r)$, $b \in H(s)$, and $a \sim b$, then $r \rightarrow_n s$;*
(iii) *$a \in H(r)$, $b \in H(s)$, and $a \sim b$, then both $r \rightarrow_p s$ and $s \rightarrow_p r$;*
(iv) *$a \in B(r)$ is an external atom of form $\&g[\mathbf{X}](\mathbf{Y})$ where $\mathbf{X} = X_1, \ldots, X_n$, the input $X_i = p$ is of type predicate, and $b \in H(s)$ is an atom of form $p(\mathbf{Z})$, then*
  - *$r \rightarrow_p s$ if $\&g$ is monotonic and $a \in B^+(r)$, and*
  - *$r \rightarrow_n s$ otherwise.*

Example 2 (ctd.). According to Definition 1, due to (i) we have dependencies $r_3 \rightarrow_p r_1$, $c_5 \rightarrow_p r_2$, and $c_5 \rightarrow_p r_3$; and due to (iv) we have dependencies $r_2 \rightarrow_p r_1$ and $r_4 \rightarrow_p r_3$. $\square$

We generically say that r *depends on* s ($r \rightarrow s$), if either $r \rightarrow_p s$ or $r \rightarrow_n s$.

Evaluation Graph. Using the above notion of rule dependencies, we define the structure of an evaluation graph consisting of evaluation units depending on one another.

We handle constraints separately from non-constraints. Constraints cannot infer additional atoms, hence they can be shared between evaluation units in many cases, while sharing non-constraints could violate the modularity of partial models. In the former evaluation algorithm, a constraint could only kill answer sets once all its dependencies were fulfilled. The new algorithm increases evaluation efficiency by duplicating nonground constraints, allowing them to kill models earlier.

In the following, an *evaluation unit* is any nonempty subset of the rules of a program. An *evaluation unit graph* is a directed graph where each vertex is a unit. Let $G = (U, E)$ be an evaluation unit graph of program P, $v \in U$, and $r \in v$. We say that the dependencies of r *are covered by G at unit v* iff for all rules s of P, if $r \rightarrow s$ holds for $s \in w$, $w \in U$, and $w \neq v$, then there is an edge from v to w in G.

Definition 2 (Evaluation graph). *An* evaluation graph *$\mathcal{E} = (V, E)$ of a program P is an acyclic evaluation unit graph such that (a) $\bigcup V = P$, (b) for each $r \in P$ and each $v \in V$ with $r \in v$, negative dependencies of r are covered by $\mathcal{E}$ at v, and (c) for each $r \in P$ its positive dependencies are covered by $\mathcal{E}$ at every (resp., some) unit $v \in V$ with $r \in v$ if r is a non-constraint (resp., constraint).*

Note that by acyclicity of $\mathcal{E}$, mutually dependent rules must be in the same unit. Furthermore, a unit can have in its rule heads only atoms which do not match atoms derivable by other units, due to dependencies between rules deriving unifiable heads.

Let $\mathcal{E} = (V, E)$ be an evaluation graph. We write $v < w$ iff there exists a path from v to w in $\mathcal{E}$, and $v \leq w$ iff either $v < w$ or $v = w$. For a unit $v \in V$, let $v^< = \bigcup_{w \in V, v < w} w$ be the set of rules in 'preceding' units on which v depends, and let

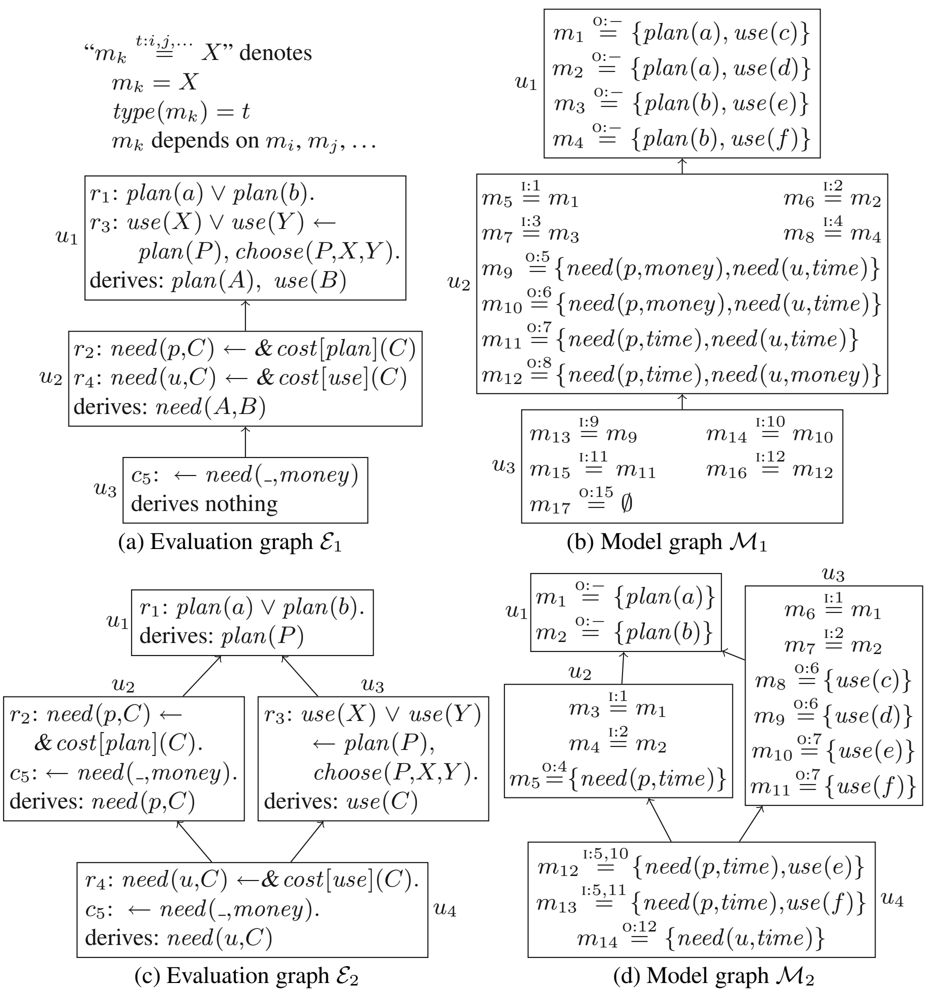

"$m_k \overset{t:i,j,\ldots}{=} X$" denotes
$m_k = X$
$type(m_k) = t$
m_k depends on $m_i, m_j, \ldots$
u_1 r_1: $plan(a) \vee plan(b)$.
r_3: $use(X) \vee use(Y) \leftarrow plan(P), choose(P,X,Y)$.
derives: $plan(A)$, $use(B)$
u_2 r_2: $need(p,C) \leftarrow \&cost[plan](C)$
r_4: $need(u,C) \leftarrow \&cost[use](C)$
derives: $need(A,B)$
u_3 c_5: $\leftarrow need(\_,money)$
derives nothing
(a) Evaluation graph $\mathcal{E}_1$
u_1 $m_1 \overset{O:-}{=} \{plan(a), use(c)\}$
$m_2 \overset{O:-}{=} \{plan(a), use(d)\}$
$m_3 \overset{O:-}{=} \{plan(b), use(e)\}$
$m_4 \overset{O:-}{=} \{plan(b), use(f)\}$
u_2 $m_5 \overset{I:1}{=} m_1$ $m_6 \overset{I:2}{=} m_2$
$m_7 \overset{I:3}{=} m_3$ $m_8 \overset{I:4}{=} m_4$
$m_9 \overset{O:5}{=} \{need(p,money), need(u,time)\}$
$m_{10} \overset{O:6}{=} \{need(p,money), need(u,time)\}$
$m_{11} \overset{O:7}{=} \{need(p,time), need(u,time)\}$
$m_{12} \overset{O:8}{=} \{need(p,time), need(u,money)\}$
u_3 $m_{13} \overset{I:9}{=} m_9$ $m_{14} \overset{I:10}{=} m_{10}$
$m_{15} \overset{I:11}{=} m_{11}$ $m_{16} \overset{I:12}{=} m_{12}$
$m_{17} \overset{O:15}{=} \emptyset$
(b) Model graph $\mathcal{M}_1$
u_1 r_1: $plan(a) \vee plan(b)$.
derives: $plan(P)$
u_2 r_2: $need(p,C) \leftarrow \&cost[plan](C)$.
c_5: $\leftarrow need(\_,money)$.
derives: $need(p,C)$
u_3 r_3: $use(X) \vee use(Y) \leftarrow plan(P), choose(P,X,Y)$.
derives: $use(C)$
u_4 r_4: $need(u,C) \leftarrow \&cost[use](C)$.
c_5: $\leftarrow need(\_,money)$.
derives: $need(u,C)$
(c) Evaluation graph $\mathcal{E}_2$
u_1 $m_1 \overset{O:-}{=} \{plan(a)\}$
$m_2 \overset{O:-}{=} \{plan(b)\}$
u_2 $m_3 \overset{I:1}{=} m_1$
$m_4 \overset{I:2}{=} m_2$
$m_5 \overset{O:4}{=} \{need(p,time)\}$
u_3 $m_6 \overset{I:1}{=} m_1$
$m_7 \overset{I:2}{=} m_2$
$m_8 \overset{O:6}{=} \{use(c)\}$
$m_9 \overset{O:6}{=} \{use(d)\}$
$m_{10} \overset{O:7}{=} \{use(e)\}$
$m_{11} \overset{O:7}{=} \{use(f)\}$
u_4 $m_{12} \overset{I:5,10}{=} \{need(p,time), use(e)\}$
$m_{13} \overset{I:5,11}{=} \{need(p,time), use(f)\}$
$m_{14} \overset{O:12}{=} \{need(u,time)\}$
(d) Model graph $\mathcal{M}_2$

Fig. 1. Old vs New strategy: evaluation and model graphs

$v^{\leq} = v^{<} \cup \{v\}$. Furthermore, for each unit $v \in V$, we define $preds_E(v) = \{w \in V \mid (v,w) \in E\}$.

Example 3 (ctd.). We focus on two specific evaluation graphs of program P. Graph $\mathcal{E}_1$ in Fig. 1a corresponds to the former HEX evaluation method. Intuitively, u_1 guesses inputs for u_2, which then evaluates rules with external atoms; finally u_3 checks constraints.

Another (possibly more efficient) evaluation graph is $\mathcal{E}_2$ in Fig. 1c: guesses of r_1 and r_3 are split into separate units u_1 and u_3, reducing redundancy of external atom evaluation. The constraint c_5 is shared between u_2 and u_4, and it prunes models in u_2, again reducing redundancy. Note that units with multiple inputs and constraint duplication do not exist in the former HEX algorithm. □

Algorithm 1. BUILDMODELGRAPH ($\mathcal{E} = (V, E)$: evaluation graph)

```
    Output: $\mathcal{M} = (M, F, unit, type)$: model graph
    $M := \emptyset, F := \emptyset, U := V$
    while $U \neq \emptyset$ do
        choose $u \in U$ s.t. $preds_E(u) \cap U = \emptyset$
        let $preds_E(u) = \{u_1, \ldots, u_k\}$ and $M' := \emptyset$
        for $m_1 \in omods_M(u_1), \ldots, m_k \in omods_M(u_k)$ do
(a)         if $m := m_1 \bowtie \cdots \bowtie m_k$ is defined then
(b)             $M' :=$ EVALUATEUNIT$(u, m)$
                set $unit(m) := u$ and $type(m) :=$ I
                set $unit(m') := u$ and $type(m') :=$ O for all $m' \in M'$
                $F := F \cup \{(m', m) \mid m' \in M'\} \cup \{(m, m_i) \mid 1 \leq i \leq k\}$
                $M := M \cup M' \cup \{m\}$
        $U := U \setminus \{u\}$
    return $(M, F, unit, type)$
```

Model Graph. We now define the model graph, which interrelates models at evaluation units. It is the foundation of our model building algorithm. In the following, a *model* m is a set of ground atoms. Each model belongs to a specific unit $unit(m)$, and has a type $type(m)$ which is either input (I) or output (O). Given a set of models M and a unit u, we denote by $imods_M(u) = \{m \in M \mid unit(m) = u,\ type(m) = \text{I}\}$ and $omods_M(u) = \{m \in M \mid unit(m) = u,\ type(m) = \text{O}\}$ the sets of input and output models of u, resp. Intuitively, when computing answer sets of a program P under an evaluation graph $\mathcal{E}$, each unit u might have a number of input models: each input model determines a particular set of input assertions for unit u, and is built in turn by merging a number of output models m_i, one per each unit $u_i \in preds_E(u)$. Given an input model m for u, the evaluation of u might 'produce' a number of output models depending on m.

Definition 3 (Model graph). *Given an evaluation graph $\mathcal{E}$=(U,E) for a program P, a* model graph *$\mathcal{M}$=$(M, F, unit, type)$ is a labelled directed acyclic graph, where each vertex m∈M is a model, $F \subseteq M \times M$, and $unit\colon M \rightarrow U$ and $type\colon M \rightarrow \{\text{I},\text{O}\}$ are vertex labelling functions. F consists of the following edges for each model m: (a) (m, m') with $m \in omods_M(u)$, $m' \in imods_M(u)$ for some $u \in U$ s.t. $preds_E(u) \neq \emptyset$; and (b) $(m, m_1), \ldots, (m, m_k)$ if $m \in imods_M(u)$, $u \in U$, and $\{u_1, \ldots, u_k\}$=$preds_E(u)$, such that m_i=$f(m, u_i) \in omods_M(u_i)$, $1 \leq i \leq k$ is some (unique) output model of u_i.*

Note that the empty graph is a model graph, and that evaluation units may have no models in the model graph. This is by intent, as our model building algorithm progresses from an empty model graph to one with models at each unit (iff the program has an answer set). Given a model m, we denote by m^+ the *expanded model* of m, which is the union of m and all output models on which m transitively depends. Note, that given m at unit u, m^+ is a union of one output model from each unit in $u^{\leq}$.

Example 4 (ctd.). In $\mathcal{M}_2$, some expanded models are $m_5^+ = \{need(p{,}time), plan(b)\}$, $m_{11}^+ = \{use(f), plan(b)\}$, and $m_{13}^+ = \{need(p{,}time), use(f), plan(b)\}$. □

4 Evaluation

Roughly speaking, answer sets of a program can be built by first obtaining an evaluation graph, and then computing a model graph accordingly. We next demonstrate model building on our example, informally discussing the main operations BUILDMODELGRAPH, model join ' $\bowtie$ ', and EVALUATEUNIT, which are later defined formally. To simplify our algorithm, we assume empty dummy input models at units without predecessors.

Example 5 (ctd.). Fig. 1b and 1d show model graphs $\mathcal{M}_1$ and $\mathcal{M}_2$ resulting from the evaluation of $\mathcal{E}_1$ and $\mathcal{E}_2$, resp. On $\mathcal{E}_1$, unit u_1 is evaluated first, yielding output models $m_1, \ldots, m_4$ containing guesses over $plan$ and use. These models are also input models for u_2; for each of them we first evaluate the external atoms $\&cost[plan](C)$ and $\&cost[use](C)$ and then evaluate $\{r_2, r_4\}$ using an external solver; we then obtain output models $m_9, \ldots, m_{12}$ which are also input models for $u_3 = \{c_5\}$. Evaluation of u_3 yields a model m_{17} for input model m_{15}, and $m_{17}^+ = \{need(p, time), need(u, time), plan(b), use(e)\}$ is the only answer set of P. For evaluation graph $\mathcal{E}_2$, we start with u_1, which yields output models m_1 and m_2. Then we process u_2 and u_3 in arbitrary order (or even in parallel). One external atom will be evaluated for each input model of u_2. Then, we evaluate $\{r_2, c_5\}$, which yields output model m_5 for input m_4, and no model for m_3. For each input model of u_3 two models are generated by r_3. Input models for u_4 are built by joining output models from u_2 and u_3 (cf. Ex. 7). Finally, u_4 evaluates one external atom per input model and then gets the models m_{14} for input m_{12} and no model for input m_{13} for $\{r_4, c_5\}$. We again get a single answer set m_{14}^+ of P. □

Model Joining. Input models of a unit u are built by combining one output model m_i for each unit u_i on which u depends. Only combinations with common ancestry in the model graph are allowed. To formalize this condition, we introduce the following notion. Unit w *is a common ancestor unit (cau) of* v in an evaluation graph $\mathcal{E} = (V, E)$ iff $v, w \in V$, $v \neq w$, and there exist distinct paths p_1, p_2 from v to w in E s.t. p_1 and p_2 overlap only in vertices v and w. We denote by $caus(v)$ the set of all caus of unit v.

Example 6. In an evaluation graph sketched by dependencies $a \to b \to c \to d \to e$, $a \to c$, and $a \to d$, we have that $caus(a) = \{c, d\}$ and no other unit besides a has caus. □

We next formally define the join operator ' $\bowtie$ ' on models.

Algorithm 2. EVALUATEUNIT(u: evaluation unit, m: input model at u)

```
Output: output models at $u$
// determine external atoms that get input only from $m$
$A_{in} := \{\&g[\mathbf{x}](\mathbf{y}) \mid r \in u$ and $x \cap \left(\bigcup_{r' \in u} H(r')\right) = \emptyset\}$
$m_{aux} := \{d_{\&g}(\mathbf{x}, \mathbf{y}) \mid \&g[\mathbf{x}](\mathbf{y}) \in A_{in}$ and $f_{\&g}(m, \mathbf{x}, \mathbf{y}) = 1\}$ // get replacement facts
$u' := u$ with external atoms $A_{in}$ replaced by their corresponding auxiliaries
choose $ES \in \{$PLAIN, WELLF, GNC$\}$ according to the structure of $u'$
return $ES(u', m \cup m_{aux})$    // return set of models for $u'$ w.r.t. $m$ and $m_{aux}$ using $ES$
```

Definition 4. *Let $\mathcal{M} = (M, F)$ be a model graph for an evaluation graph $\mathcal{E} = (V, E)$ of a program P, and let $u \in V$ be a unit. Let $u_1, \ldots, u_k$ be all the units u_i on which u depends, and let $m_i \in omods_M(u_i)$, $1 \leq i \leq k$. Then the* join $m = m_1 \bowtie \cdots \bowtie m_k = \bigcup_{1 \leq i \leq k} m_i$ *is defined iff for each $u' \in caus(u)$ there exists exactly one model $m' \in omods_M(u')$ reachable (in $\mathcal{M}$) from some model m_i, $1 \leq i \leq k$.*

Example 7 (ctd.). Building input models for u_4 in Figure 1d requires a join operation: from all pairs of output models at u_2 and u_3, only those with a common ancestor at u_1 yield a valid input model: u_4 has two input models $m_5 \bowtie m_{10}$ and $m_5 \bowtie m_{11}$; they have m_2 as a common ancestor at u_1. For other combinations, the join is undefined. □

Evaluation Algorithm. Alg. 1 builds our model graph: U contains units for which models still have to be calculated; in each iteration step (a) determines all input models m for unit u, step (b) calculates output models originating in m.

EVALUATEUNIT (Alg. 2) evaluates unit u; it creates output models for given input model m. Given a possibly non-ground external atom $\&g[\mathbf{x}](\mathbf{y})$, we denote by the ordinary atom $d_{\&g}(\mathbf{x}, \mathbf{y})$ its *corresponding replacement atom*. These replacement atoms are instrumental for evaluating rules containing external atoms; we apply the approach of "HEX component evaluation" introduced in [20]: intuitively, we evaluate external atom functions wrt. a given input model m, augment m with replacement facts for inputs where $f_{\&g}$ evaluates to 1, replace external atoms by corresponding replacement atoms in all rule bodies, and then evaluate the resulting program R wrt. augmented input model m'. Depending on the structure of R, we can choose between different *evaluation strategies* as described in [20]; PLAIN: if R contains no external atoms, we create output models $\mathcal{AS}(R \cup m')$ by an external solver; WELLF: if external atoms in R are monotonic, and none is contained in a negative dependency cycle, we use a fixpoint algorithm; and GNC: in all other cases, we use a guess-and-check algorithm.

Soundness and Completeness. Because of constraint duplication, the evaluation graph does not partition the input program, and the customary notion of splitting set does not apply to evaluation units. We define a *generalized bottom* of a program, that is a way to split a program into two parts with a nonempty intersection containing certain constraints. We prove that generalized bottoms behave similar as bottoms gb_A created by global splitting sets and $EVAL$ [7], to which we only refer here.

Definition 5. *Given a* HEX *program P, a* generalized bottom *$P' \subseteq P$ is a subset of P such that there exists a global splitting set A and the set $C = P' \setminus gb_A(P)$ is a set of constraints with $B^-(C) \subseteq A$.*

Definition 6 (as in [7]). *For an interpretation I and a program Q, the* global residual, *$gres(Q, I)$, is the program obtained from Q as follows: (i) add all the atoms in I as facts, and (ii) for each "resolved" external atom $a = \&g[\mathbf{X}](\mathbf{Y})$ occurring in some rule of Q, replace a with a fresh ordinary atom $d_{\&g}(\mathbf{c})$ for each tuple $\mathbf{c}$ output by $EVAL(\&g, Q, I)$.*

Theorem 1. *Let P be a domain-expansion safe* HEX *program over atoms U, and let P' be a generalized bottom of P with global splitting set A and constraints C. Then $M \setminus D \in \mathcal{AS}(P)$ iff $M \in \mathcal{AS}(gres(P'', I))$ with $I \in \mathcal{AS}(P')$ and $P'' = (P \setminus P') \cup C'$,* *where D is the set of additional atoms in $gres(P'')$ with predicate name of form $d_{\&g}$, and $C' = \{c \in C \mid B^+(c) \cap (U \setminus A) \neq \emptyset\}$ is the set of constraints in P' with body atoms unifying with atoms in A as well as with atoms in $U \setminus A$.*

Intuitively, this is a relaxation of the previous nonground HEX splitting theorem regarding constraints: those matching atoms derived in the splitting set as well as in the residual program may be added to P' iff they are not removed from the residual program. The benefit of sharing such constraints is a reduced set of models $\mathcal{AS}(P')$.

The following lemma applies the above splitting theorem to the evaluation graph and the model graph, and is instrumental for showing correctness of the algorithm.

Lemma 1. *Given an evaluation graph $\mathcal{E} = (V_E, E_E)$ of a* HEX *program P, and an evaluation unit $u \in V_E$, it holds that (i) the subprogram $u^<$ is a generalized bottom of the subprogram $u^\leq$; furthermore if for each predecessor $u' \in preds_E(u)$ we have that models $\{m'^+ \mid m' \in omods_M(u')\}$ are the models of $u'^\leq$, it holds that (ii) step (a) of* BUILDMODELGRAPH *creates the models of bottom $u^<$ as $imods_M(u)$, and (iii) step (b) builds models $omods_M(u)$ of u s.t. $\{m^+ \mid m \in omods_M(u)\}$ are the models of $u^\leq$.*

Using this lemma, we can inductively prove that each iteration of BUILDMODELGRAPH chooses a unit u without models, creates input models and then output models at u, such that all expanded models at unit u are answer sets of subprogram $u^\leq$. In this manner, the model graph is extended until no longer possible. We have the following result.

Theorem 2. *Given an evaluation graph $\mathcal{E} = (V, E)$ of a* HEX *program P,* BUILDMODELGRAPH *returns the model graph $\mathcal{M} = (M, F)$ such that $\{m_1 \bowtie \cdots \bowtie m_n \mid m_i \in omods_M(u_i),\ u_i \in V\} = \mathcal{AS}(P)$.*

5 Implementation and Experiments

The presented framework has been implemented to become the next version of the dlvhex solver: dlvhex 2.0 (`http://www.kr.tuwien.ac.at/research/systems/dlvhex/`). The current implementation supports dlv (`http://www.dlvsystem.com/`) and (for a non-disjunctive fragment of HEX) clasp+gringo (`http://potassco.sourceforge.net/`) as back-end ASP solvers.

In addition to the framework described above, an online model calculation algorithm has been implemented that can easily be extended to add query support. So far, the evaluation strategy PLAIN has been implemented; implementing other strategies just requires adapting legacy code to new C++ data structures. Two evaluation heuristics are implemented: the former dlvhex evaluation heuristics *H1* and a new heuristics *H2* (cf. Ex. 3). *H1* was ported for comparing dlvhex 1.x to 2.x; it splits a given program into strongly connected components and external components (which are as small as possible). The new *H2* places rules into units as follows: (i) combine rules r_1, r_2 whenever $r_1 \rightarrow s$ and $r_2 \rightarrow s$ and there is no rule t s.t. exactly one of r_1, r_2 depends on t; (ii) combine rules r_1, r_2 whenever $s \rightarrow r_1$ and $s \rightarrow r_2$ and there is no rule t s.t. t depends on exactly one of r_1, r_2; but (iii) never combine rules r, s if r contains external atoms and $r \rightarrow s$. Intuitively, *H2* builds an evaluation graph that puts all rules with external atoms and their successors into one unit, while separating rules creating input for distinct external atoms. This avoids redundant computation and joining unrelated models.

Experimental Setup and Benchmarks. A series of 6 concurrent tests were run on a Linux machine with two quad-core Intel Xeon 3GHz CPUs and 32GB RAM. The system resources were limited to a maximum of 3GB memory usage and 600 secs execution time for each run. The computation task for all experiments was to compute all answer sets of two kinds of benchmark instances:

• MCS. The first kind of benchmark instances, motivating this research, are HEX programs capturing multi-context systems (MCS)—a formalism for interlinking distributed knowledge sources with possibly nonmonotonic "bridge rules" (see [2]). Each instance consists of 5–10 guessed atoms of input and output interpretations for each of 7–9 knowledge sources, which are realized by external atoms in constraints. Most guesses are eliminated by these constraints, the remaining guesses are linked by HEX rules representing bridge rules of the modeled system. These benchmarks come in 14 different flavors (bridge rule topologies and sizes), each with 10 randomized single instances. Instances have an average of 400 models, with values ranging from 4 to $\sim$20,000 models.
• REVIEWER SELECTION (REVSEL). The second class of benchmark instances encode the selection of reviewers for conference papers—taking conflicts into account, some of which are encoded by external atoms. For these instances, we vary the number T of conference tracks and the number P of papers per track. The number of reviewers available for each track equals P and there is one reviewer assigned to all tracks (establishing a dependency between conference track assignments). Each paper must have 2 reviews and no reviewer gets more than 2 papers assigned. We generated conflicts such that we limit the number of overall models, as well as the number of candidate models per conference track, before checking conflicts modeled via external atoms.

We consider two special classes of reviewer selection. In REVSEL 1, we first compared the old and the new evaluation approach for a very specific program structure, as well as the old and new implementation with the ported (old) evaluation heuristics *H1*. For that we used $P = 20$ papers per conference track and varied the number of tracks T. External atoms and conflicts were configured such that all conference tracks have two solutions before evaluating constraints with external atoms, and one overall model after program evaluation. The REVSEL 2 experiment involved no external atoms: we used $T = 5$ conference tracks and varied the number of papers per track. Conflicts are generated such that there are 1-2 solutions per conference track, with a shared reviewer such that each program Q has 9 answer sets in total.

Results. It turned out that on the considered problems, the new evaluation approach outperforms the old one significantly, using less memory (sometimes exponentially less). For the MCS benchmark instances, the old approach had 34 timeouts and 83 memory outages; thus only 16% of all instances triggered some output. The average time and memory usage for successful termination was 86 seconds and 623MB, resp. Both values have a high standard deviation. In contrast, the new approach successfully calculated all models for all instances with an average solve time of 3 seconds and an average memory usage of 32MB, both with a small standard deviation. This big improvement makes usage of HEX programs in this problem domain feasible for the first time. Note that this

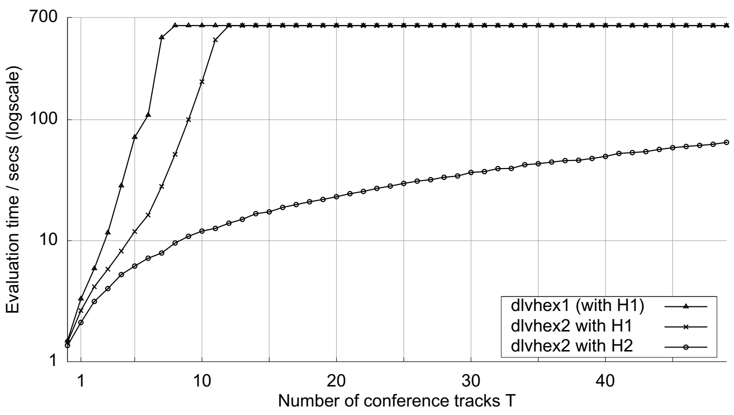


Fig. 2. RevSel 1 ($P = 20$), out-of-memory for $T \geq 12$

problem domain was originally not generated for benchmarking HEX programs, so it is not specifically geared towards showing beneficial effect of our new approach.

Results for RevSel 1 are shown in Fig. 2: an exponential increase of runtime is visible in the old approach, compared to linear time growth of the new one. Memory usage behaves similarly. In general, increasing T causes timeouts, yet bigger T's exhaust memory. Under *H1*, dlvhex 2 acts better, which may be explained with technical improvements. As a surprising result of RevSel 2, our divide-and-conquer approach performs better than solving Q directly with native solvers. This has been observed for both dlv and clasp as a backend. Our prototype incurs a small overhead for decomposing the program: small instances with running time < 2 sec are slower than native solvers, while big instances using *H2* were solved faster and with significantly less memory usage.

6 Discussion and Conclusion

We illustrated a new general technique for evaluating nonmonotonic logic programs with external sources of computation. Our work is clearly related to work on program modularity under stable model semantics, including, e.g., the seminal paper [13] on splitting sets, and [15,12], which lifted them to modular programs with choice rules and disjunctive rules and allow for "symmetric splitting." An important difference is that our decomposition approach works for nonground programs and explicitly considers the possibility that modules overlap. It is tailored to efficient evaluation of arbitrary programs, rather than to facilitate module-style logic programming with declarative specifications. In this regard, it is in line with previous work on HEX program evaluation [7] and decomposition techniques for efficient grounding of ordinary programs [3].

The work presented here can be furthered in different directions. As for the prototype reasoner, a rather straightforward extension is to support brave and cautious reasoning on top of HEX programs, while incorporating constructs like aggregates or preference constraints requires more care and efforts. Regarding program evaluation, our general evaluation framework provides a basis for further optimizations that, as indicated by our experiments, are also of interest for ordinary logic programs. Indeed, the generic notions of evaluation unit, evaluation plan and model graph allow to specialize and improve our framework in different respects: first, evaluation units (which may contain duplicated constraints), can be chosen according to a proper estimate of the number of answer sets (the fewer, the better); second, evaluation plans can be chosen by ad-hoc optimization modules, which may give preference to time, space, or parallelization requirements, or to a combination of the three. Furthermore, our framework is ready to a form of coarse-grained distributed computation at the level of evaluation units (in the style of [16]): evaluation graphs naturally encode parallel evaluation plans. Independent units can in fact be evaluated in parallel, while our 'model streaming' architecture lends itself to pipelined evaluation of subsequent modules. Improving reasoning performance by decomposition has been investigated in [1], however, only wrt. monotonic logics.

As a last remark on possible optimizations, we observe that the data flow (constituted by intermediate answer sets) between evaluation units can be optimized using proper notions of model projection, such as in [9]. Model projections would tailor input data of evaluation units to necessary parts of intermediate answer sets; however, given that different units might need different parts of the same intermediate input answer set, a space-saving efficient projection technique is not straightforward.

References

1. Amir, E., McIlraith, S.: Partition-based logical reasoning for first-order and propositional theories. Artif. Intell. 162(1-2), 49–88 (2005)
2. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI 2007, pp. 385–390. AAAI Press, Menlo Park (2007)
3. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 407–424. Springer, Heidelberg (2008)
4. Eiter, T., Fink, M., Krennwallner, T.: Decomposition of Declarative Knowledge Bases with External Functions. In: IJCAI 2009, pp. 752–758. AAAI Press, Menlo Park (2009)
5. Eiter, T., Ianni, G., Krennwallner, T., Schindlauer, R.: Exploiting conjunctive queries in description logic programs. Ann. Math. Artif. Intell. 53(1-4), 115–152 (2008)
6. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: IJCAI 2005, pp. 90–96 (2005)
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective integration of declarative rules with external evaluations for semantic-web reasoning. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 273–287. Springer, Heidelberg (2006)
8. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
9. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 71–86. Springer, Heidelberg (2009)
10. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. Next Generat. Comput. 9(3-4), 365–386 (1991)

11. Hoehndorf, R., Loebe, F., Kelso, J., Herre, H.: Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. BMC Bioinf. 8(1), 377 (2007), doi:10.1186/1471-2105-8-377
12. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. J. Artif. Intell. Res. 35, 813–857 (2009), doi:10.1613/jair.2810
13. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: ICLP 1994, pp. 23–38. MIT Press, Cambridge (1994)
14. Van Nieuwenborgh, D., Eiter, T., Hadavandi, E.: Conditional planning with external functions. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 214–227. Springer, Heidelberg (2007)
15. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. Theory Pract. Log. Prog. 8(5-6), 717–761 (2008)
16. Perri, S., Ricca, F., Sirianni, M.: A parallel ASP instantiator based on DLV. In: DAMP 2010, pp. 73–82. Springer, Heidelberg (2010)
17. Polleres, A.: From SPARQL to rules (and back). In: WWW 2007, pp. 787–796. ACM, New York (2007)
18. Przymusinski, T.C.: On the Declarative Semantics of Deductive Databases and Logic Programs. In: Foundations of Deductive Databases and Logic Programming, pp. 193–216 (1988)
19. Ross, K.: Modular Stratification and Magic Sets for Datalog Programs with Negation. J. ACM 41(6), 1216–1267 (1994)
20. Schindlauer, R.: Answer-set programming for the Semantic Web. Ph.D. thesis, Vienna University of Technology (2006)
21. Zirtiloğlu, H., Yolum, P.: Ranking semantic information for e-government: complaints management. In: OBI 2008, vol. (5), pp. 1–7. ACM, New York (2008), doi:10.1145/1452567.1452572

Approximations for Explanations of Inconsistency in Partially Known Multi-Context Systems[*]

Thomas Eiter, Michael Fink, and Peter Schüller

Institute of Information Systems
Vienna University of Technology
Favoritenstrasse 11, A-1040 Vienna, Austria
{eiter,fink,schueller}@kr.tuwien.ac.at

Abstract. Multi-context systems are a formalism to interlink decentralized and heterogeneous knowledge based systems (contexts), which interact via possibly nonmonotonic bridge rules. Inconsistency is a major problem, as it renders such systems useless. In applications involving confidentiality or trust, it is likely that complete knowledge about all system parts is unavailable. To address inconsistency in such scenarios, we extend existing notions for characterizing inconsistency in multi-context systems: we propose a representation of partial knowledge, and introduce a formalism for approximating reasons of inconsistency. We also discuss query selection strategies for improving approximations in situations where a limited number of queries can be posed to a partially known context.

1 Introduction

In recent years, there has been an increasing interest in interlinking knowledge bases, in order to enhance the capabilities of systems. Based on McCarthy's idea of contextual reasoning [11], the Trento School around Giunchiglia and Serafini has developed multi-context systems in many works, in which the components (called contexts) can be interlinked via so called bridge rules for information exchange, cf. [9,5]. Generalizing this work, Brewka and Eiter [4] presented nonmonotonic multi-context systems (MCSs) as a generic framework for interlinking possibly heterogeneous and nonmonotonic knowledge bases.

Typically, an MCS is not built from scratch, but assembled from components which were not specifically designed to be part of a more complex system. Unintended interactions between contexts thus may easily arise and cause inconsistency, which renders an MCS useless. Making bridge rules defeasible [2] avoids inconsistency and cures faults in silent service. However, underlying reasons for inconsistency may remain unnoticed and cause unpleasant side-effects that are difficult to track.

Therefore, to help the user analyze, understand and eventually repair inconsistencies, suitable notions of consistency-based diagnosis and entailment-based explanation for inconsistency were introduced in [8]. Intuitively, diagnoses represent possible system repairs, while explanations characterize sources of inconsistency. An omniscient view of the system was assumed, where the user has full information about all contexts

[*] Supported by the Vienna Science and Technology Fund (WWTF) under grant ICT08-020.

including their knowledge bases and semantics. However, in many real world scenarios full information is not available [3], and some contexts are black boxes with internal knowledge bases or semantics that are not disclosed due to intellectual property or privacy issues (e.g., banks will not disclose their full databases to credit card companies). Partial behavior of such contexts may be known, however querying might be limited, e.g., by contracts or costs. In such scenarios, inconsistencies can only be explained given the knowledge of the system one has, and since this is partial, the explanations obtained just approximate the actual situation, i.e., those explanations one would obtain if one would have full insight.

In other words, this calls for explaining inconsistency in an MCS with *partial knowledge* about contexts, which raises the following technical challenges:

- how to represent partial knowledge about the system, and
- how to obtain reasonable *approximations* forexplanations of inconsistency in the actual system (under full knowledge), ideally in an efficient way.

The first issue depends on the nature of this knowledge, and a range of possibilities exists. The second issue requires an assessment methodto determine such approximations. We tackle both issues and make the following contributions.

• We develop a representation of partially known contexts, which is based on context abstraction with Boolean functions. Partially defined Boolean functions [15,7] are used to capture partially known behavior of a context.
• We exploit these representations to define *over-* and *underapproximations* of diagnoses and explanations for inconsistency according to [8], in the presence of partially known contexts. The approximations target either the whole set of diagnoses, or one diagnosis at a time; analogously for explanations.
• For scenarios where partially known contexts can be asked a limited number of queries, we consider query selection strategies.
• Finally, we discuss computational complexity of recognizing approximate explanations. In contrast to semantic approximations for efficient evaluation [12], our approximations handle incompleteness, which usually increases complexity. Fortunately, our approach does not incur higher computational cost than the case of full information.

Our results extend methods for inconsistency handling in MCSs to more realistic settings, e.g., in health-care where privacy issues need to be respected, without increasing computational cost. In practical applications our approximations reduce the set of system parts relevant for restoring consistency, and allow for better focussing of repair efforts.

2 Preliminaries

A heterogeneous nonmonotonic MCS [4] consists of *contexts*, each composed of a knowledge base with an underlying *logic*, and a set of *bridge rules*

A logic $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$ is an abstraction, which allows to capture many monotonic and nonmonotonic logics, e.g., classical logic, description logics, default logics, etc. It consists of the following components:

- $\mathbf{KB}_L$ is the set of well-formed knowledge bases of L. We assume each element of $\mathbf{KB}_L$ is a set of "formulas".
- $\mathbf{BS}_L$ is the set of possible belief sets, where a belief set is a set of "beliefs".
- $\mathbf{ACC}_L : \mathbf{KB}_L \rightarrow 2^{\mathbf{BS}_L}$ is a function describing the semantics of the logic by assigning to each knowledge base a set of acceptable belief sets.

Each context has its own logic, which allows to model heterogeneous systems.

A *bridge rule* models information flow between contexts: it can add information to a context, depending on the belief sets accepted at other contexts. Let $L = (L_1, \dots, L_n)$ be a tuple of logics. An L_k-bridge rule r over L is of the form

$$(k : s) \leftarrow (c_1 : p_1), \dots, (c_j : p_j), \mathbf{not}\ (c_{j+1} : p_{j+1}), \dots, \mathbf{not}\ (c_m : p_m). \quad (1)$$

where $1 \leq c_i \leq n$, p_i is an element of some belief set of L_{c_i}, and k refers to the context receiving formula s. We denote by $hd\ (r)$ the formula s in the head of r.

Definition 1. *A multi-context system $M = (C_1, \dots, C_n)$ is a collection of contexts $C_i = (L_i, kb_i, br_i)$, $1 \leq i \leq n$, where $L_i = (\mathbf{KB}_i, \mathbf{BS}_i, \mathbf{ACC}_i)$ is a logic, $kb_i \in \mathbf{KB}_i$ a knowledge base, and br_i is a set of L_i-bridge rules over $(L_1, \dots, L_n)$. By $IN_i = \{hd\ (r) \mid r \in br_i\}$ we denote the set of possible* inputs *of context C_i added by bridge rules, and by $br_M = \bigcup_{i=1}^{n} br_i$ the set of all bridge rules of M.*

In addition, for each $H \subseteq IN_i$ we must have $kb_i \cup H \in \mathbf{KB}_{L_i}$.

The following running example involves policies and trust information which are often non-public and distributed [3], and thus demonstrates the necessity of reasoning under incomplete information. For more examples of MCSs see [4,8].

Example 1. Consider an MCS M consisting of a permission database $C_1 = C_{\mathbf{perm}}$ and a credit card clearing context $C_2 = C_{\mathbf{cc}}$, and the following bridge rules:

$$\begin{aligned}
r_1 : (\mathbf{perm} : \mathit{person}(\mathit{Person})) &\leftarrow \top. \\
r_2 : (\mathbf{cc} : \mathit{card}(\mathit{CreditCard})) &\leftarrow (\mathbf{perm} : \mathit{person}(\mathit{Person})), \\
&\quad\ \mathbf{not}\ (\mathbf{perm} : \mathit{grant}(\mathit{Person})), \\
&\quad\ (\mathbf{perm} : \mathit{ccard}(\mathit{Person}, \mathit{CreditCard})). \\
r_3 : (\mathbf{perm} : \mathit{ccValid}(\mathit{CreditCard})) &\leftarrow (\mathbf{cc} : \mathit{valid}(\mathit{CreditCard})).
\end{aligned}$$

Here r_1 defines a set of persons which is relevant for permission evaluation in $C_{\mathbf{perm}}$; r_2 specifies that, if some person is not granted access, credit cards of that person have to be checked; and r_3 translates validation results to $C_{\mathbf{perm}}$.

The MCS formalism is defined on ground bridge rules, which are in the following denoted by $r_{i,<constants>}$, e.g., $r_{2,moe,cnr2}$ denotes r_2 with $\mathit{Person} \mapsto \mathit{moe}$ and $\mathit{CreditCard} \mapsto \mathit{cnr2}$. Unless stated otherwise, we assume that bridge rules are grounded with $\mathit{Person} \in \{\mathit{nina}, \mathit{moe}\}$ and $\mathit{CreditCard} \in \{\mathit{cnr1}, \mathit{cnr2}\}$.

We next describe the context internals: $C_{\mathbf{perm}}$ is a datalog program with the following logic: $\mathbf{KB}_{\mathbf{perm}}$ contains all syntactically correct datalog programs, $\mathbf{BS}_{\mathbf{perm}}$ contains all possible answer sets, and $\mathbf{ACC}_{\mathbf{perm}}$ returns for each datalog program the corresponding answer sets. The knowledge base $kb_{\mathbf{perm}}$ is as follows:

$$\begin{array}{lll} \mathit{group}(\mathit{nina}, \mathit{vip}). & \mathit{ccard}(\mathit{nina}, \mathit{cnr1}). & \mathit{ccard}(\mathit{moe}, \mathit{cnr2}). \\ \mathit{igrant}(\mathit{Person}) & \leftarrow \mathit{person}(\mathit{Person}), \mathit{group}(\mathit{Person}, \mathit{vip}). & \\ \mathit{grant}(\mathit{Person}) & \leftarrow \mathit{igrant}(\mathit{Person}). & \\ \mathit{grant}(\mathit{Person}) & \leftarrow \mathit{ccValid}(\mathit{CreditCard}), \mathit{ccard}(\mathit{Person}, \mathit{CreditCard}). & \end{array}$$

Context $C_{\mathbf{cc}}$ is a credit card clearing facility, which typically is neither fully disclosed to the operator, nor can it be queried without significant cost. Hence, one obviously has to deal with partial knowledge: $C_{\mathbf{cc}}$ accepts $\mathit{valid}(X)$ iff card X is valid and validation is requested by $\mathit{card}(X)$. Without full insight or a history of past requests, weonly know the behavior of $C_{\mathbf{cc}}$ when no bridge rules are applicable: $\mathbf{ACC_{cc}}(kb_{\mathbf{cc}} \cup \emptyset) = \{\emptyset\}$. □

Equilibrium semantics selects certain belief states of an MCS $M = (C_1, \ldots, C_n)$ as acceptable. A *belief state* is a sequence $S = (S_1, \ldots, S_n)$, s.t. $S_i \in \mathbf{BS}_i$. A bridge rule (1) is *applicable* in S iff for $1 \leq i \leq j$: $p_i \in S_{c_i}$ and for $j < l \leq m$: $p_l \notin S_{c_l}$. Let $app(R, S)$ denote the set of bridge rules in R that are applicable in belief state S.

Intuitively, an equilibrium is a belief state S, where each context C_i takes into account the heads of all bridge rules that are applicable in S, and accepts S_i.

Definition 2. *A belief state $S = (S_1, \ldots, S_n)$ of M is an equilibrium iff, for $1 \leq i \leq n$, the following condition holds: $S_i \in \mathbf{ACC}_i(kb_i \cup \{hd\,(r) \mid r \in app(br_i, S)\})$. By* $\mathrm{EQ}(M)$ *we denote the set of equilibria of M.*

Example 2 (ctd). Assume that M_1 is the MCS M with just $\mathit{person}(\mathit{nina})$ present at $C_{\mathbf{perm}}$. As nina is in the vip group there is no need to verify a credit card, and M_1 has the following equilibrium (we omit facts, that are present in $kb_{\mathbf{perm}}$): $(\{\mathit{person}(\mathit{nina}), \mathit{igrant}(\mathit{nina}), \mathit{grant}(\mathit{nina})\}, \emptyset)$. □

Inconsistency in an MCS is the lack of an equilibrium. No information can be obtained from an inconsistent MCS, i.e., reasoning tasks on equilibria become trivial. Therefore we analyze inconsistency in order to explain and eventually repair it.

Explanation of Inconsistency. We use the notions of consistency-based *diagnosis* and entailment-based *inconsistency explanation* in MCSs [8], which aim at describing inconsistency by sets of involved bridge rules.

Given an MCS M and a set R of bridge rules, by $M[R]$ we denote the MCS obtained from M by replacing its set of bridge rules br_M with R (in particular, $M[br_M] = M$ and $M[\emptyset]$ is M with no bridge rules). By $M \models \bot$ we denote that M is inconsistent, i.e., $\mathrm{EQ}(M) = \emptyset$, and by $M \not\models \bot$ the opposite. For any set of bridge rules A, $\mathit{heads}(A) = \{\alpha \leftarrow \mid \alpha \leftarrow \beta \in A\}$ are the rules in A in unconditional form. For pairs $A = (A_1, A_2)$ and $B = (B_1, B_2)$ of sets, the pointwise subset relation $A \subseteq B$ holds iff $A_1 \subseteq B_1$ and $A_2 \subseteq B_2$. We denote by $S|_A$ the projection of all sets X in set S to set A, formally $S|_A = \{X \cap A \mid X \in S\}$.

Definition 3. *Given an MCS M, a* diagnosis *of M is a pair (D_1, D_2), $D_1, D_2 \subseteq br_M$, s.t. $M[br_M \setminus D_1 \cup \mathit{heads}(D_2)] \not\models \bot$. $D^{\pm}(M)$ is the set of all such diagnoses. $D^{\pm}_m(M)$ is the set of all pointwise subset-minimal diagnoses of an MCS M.*

A diagnosis points out bridge rules which need to be modified to restore consistency; each rule can either be deactivated, or added unconditionally. Diagnoses represent concrete system repairs by these two basic actions, but thus also characterize more sophisticated ways of repair [8]. Moreover, weassume that context knowledge bases are consistent if no bridge rule heads are added (i.e., $\forall C_i : \mathbf{ACC}_i(kb_i) \neq \emptyset$) so restoring consistency is always possible (by removing all bridge rules). For more background and discussion of this notion, we refer to [8]; for inconsistency explanations cf. also Definition 9. We next give an example of an inconsistent MCS and its diagnoses.

Example 3 (ctd). Let M_2 be the MCS M with just $person(moe)$ present at $C_{\mathbf{perm}}$, and assume the following full knowledge about $C_{\mathbf{cc}}$: all credit cards are valid.

M_2 is inconsistent: *moe* is not in the *vip* group, card verification is required by $r_{2,moe,cnr2}$, and $C_{\mathbf{cc}}$ accepts $valid(cnr2)$. This allows $C_{\mathbf{perm}}$ to derive $grant(moe)$, which blocks applicability of $r_{2,moe,cnr2}$. Therefore, M_2 contains an unstable cycle and is inconsistent. Two $\subseteq$-minimal diagnoses of M_2 are then as follows: $(\{r_{2,moe,cnr2}\}, \emptyset)$ (do not validate *cnr2*), and $(\emptyset, \{r_{2,moe,cnr2}\})$ (always validate *cnr2*)[1]. This points out r_2 as a likely culprit of inconsistency. Indeed, r_2 should intuitively contain $igrant(Person)$ in its body instead of $grant(Person)$. □

In this work we develop an approach which is able to point out a problem in r_2, without requiring complete knowledge.

3 Information Hiding

In this section, we introduce an abstraction of contexts which allows us to calculate diagnoses and explanations. We generalize this abstraction to represent partial knowledge, i.e., contexts C_i where either kb_i, or $\mathbf{ACC}_i$ is only partially known.

Context Abstraction. We abstract from a context's knowledge base kb_i and logic L_i by a Boolean function over the context's inputs IN_i (see Definition 1) and over the context's *output beliefs* OUT_i, which are those beliefs p in $\mathbf{BS}_i$ that occur in some bridge rule body in br_M as "$(i{:}p)$" or as "$\mathbf{not}\ (i{:}p)$" (see also [8]).

Recall that a Boolean function (BF) is a map $f : \mathbb{B}^k \to \mathbb{B}$ where $k \in \mathbb{N}$ and $\mathbb{B} = \{0, 1\}$. Such a BF can also be characterized either by its true points $T(f) = \{\vec{x} \mid f(\vec{x}) = 1\}$, or by its false points $F(f) = \{\vec{x} \mid f(\vec{x}) = 0\}$.

Given a set $X \subseteq U = \{u_1, \ldots, u_k\}$, we denote by $\vec{x}_U$ the characteristic vector of X wrt. some universe U (i.e. $\vec{x}_U = (b_1, \ldots, b_k)$, where $b_i = 1$ if $u_i \in X$, 0 otherwise). If understood, we omit U. Using this notation, we characterize sets of bridge rule heads $I \subseteq IN_i$ and sets of output beliefs $O \subseteq OUT_i$ by vectors $\vec{\imath}_{IN_i}$ and $\vec{o}_{OUT_i}$, respectively. For example, given $O = \{a, c\}$, and $OUT_i = \{a, b, c\}$, we have $\vec{o} = (1, 0, 1)$.

Example 4 (ctd). We use the following (ordered) sets for inputs and output beliefs: $IN_{\mathbf{cc}} = \{card(cnr1), card(cnr2)\}$, and $OUT_{\mathbf{cc}} = \{valid(cnr1), valid(cnr2)\}$. □

Definition 4. *The unique BF $f^{C_i} : \mathbb{B}^{|IN_i|+|OUT_i|} \to \mathbb{B}$ corresponds to the semantics of context C_i in an MCS M as follows:*

$$\forall I \subseteq IN_i, O \subseteq OUT_i : f^{C_i}(\vec{\imath}, \vec{o}) = 1 \textit{ iff } O \in \mathbf{ACC}_i(kb_i \cup I)\big|_{OUT_i}.$$

[1] Other $\subseteq$-minimal diagnoses of M_2 are $(\{r_{1,moe}\}, \emptyset)$, $(\{r_{3,cnr2}\}, \emptyset)$, and $(\emptyset, \{r_{3,cnr2}\})$.

Example 5 (ctd). With full knowledge (see Example 3), $C_{\mathbf{cc}}$ has as corresponding BF the function $f^{C_{\mathbf{cc}}}(X,Y,X,Y) = 1$ for all $X, Y \in \mathbb{B}$, 0 otherwise. □

If a context accepts a belief set O' for a given input I, we obtain the true point $(\vec{\mathrm{I}}, \vec{\mathrm{O}})$ of f with $O = O' \cap OUT_i$. Similarly, each non-accepted belief set yields a false point of f. Due to projection, different accepted belief sets can characterize the same true point.

Consistency Checking. Context abstraction provides sufficient information to calculate *output-projected equilibria* of the given MCS. Hence, it also allows for checking consistency and calculating diagnoses and explanations.

Given a belief state $S = (S_1, \dots, S_n)$ in MCS M, the *output-projected belief state* $S' = (S'_1, \dots, S'_n)$, $S'_i = S_i \cap OUT_i$, $1 \leq i \leq n$, is the projection of S to the output beliefs of M. In the following, we implicitly use the prime "$'$" to denote output-projection.

Definition 5 (see also [8]). *An output-projected belief state* $S' = (S'_1, \dots, S'_n)$ *of an MCS* M *is an* output-projected equilibrium *iff, for* $1 \leq i \leq n$, *it holds that* $S'_i \in \mathbf{ACC}_i(kb_i \cup \{hd\,(r) \mid r \in app(br_i, S')\})\big|_{OUT_i}$.
By $\mathrm{EQ}'(M)$ *we denote the set of output-projected equilibria of* M.

Since $app(br_i, S) = app(br_i, S')$, a simple consequence is:

Lemma 1 ([8]). *For each equilibrium* S *of an MCS* M, S' *is an output-projected equilibrium; conversely, for each output-projected equilibrium* S' *of* M *there exists at least one equilibrium* T *of* M *such that* $T' = S'$.

This means, that output-projected equilibria provide precise characterizations of equilibria on beliefs which are relevant for bridge rule applicability, i.e., on output beliefs, but are indifferent on all other beliefs. The representation of a context by a BF provides an input/output oracle, projected to output beliefs. Therefore, the BF is sufficient for consistency checking as well.

Thus, towards a representation of an MCS with partial knowledge of certain contexts, we next provide a notation for an MCS M where the knowledge of a context C_i is given by BF f, rather than kb_i.

Definition 6. *Given MCS* $M = (C_1, \dots, C_n)$, *BF* f *and index* $1 \leq i \leq n$. *We denote by* $M[i/f]$ *the MCS* M *where context* C_i *is replaced by a context* $C(f)$ *which contains the set* br_i *of bridge rules, a logic with a signature that contains* $IN_i \cup OUT_i$, *and* $kb_{C(f)}$ *and* $\mathbf{ACC}_{C(f)}$, *such that* $f^{C(f)} = f^{C_i}$.

For instance, $C(f)$ could be based on classical logic or logic programming, with $kb_{C(f)}$ over $IN \cup OUT$ as atoms encoding f by clauses (rules) that realize the correspondence.

We now show that a BF representation of a context is sufficient for calculating output-projected equilibria. We denote by $M[i_1, \dots, i_k/f_1, \dots, f_k]$ the substitution of pairwise distinct contexts $C_{i_1}, \dots, C_{i_k}$ by $C(f_1), \dots, C(f_k)$, respectively.

Theorem 1. *Let* $M = (C_1, \dots, C_n)$ *be an MCS, and let* $f_{i_1}, \dots, f_{i_k}$ *be BFs that correspond to* $C_{i_1}, \dots, C_{i_k}$. *Then,* $\mathrm{EQ}'(M) = \mathrm{EQ}'(M[i_1, \dots, i_k/f_{i_1}, \dots, f_{i_k}])$.

Partially Known Contexts. As the BF representation concerns only output beliefs, it already hides part of the context, while we are still able to analyze inconsistency. Now we generalize the BF representation to *partially defined Boolean functions* (pdBFs) (cf. [15,7]), to represent contexts where we have only partial knowledge about their output-projected behavior.

In applications, existence of such partial knowledge is realistic: for some bridge rule firings one may know an accepted belief set of a context, but not whether other accepted belief sets exist. Similarly one may know that a context is inconsistent for some input combination, but not whether it accepts some belief set for other input combinations.

Formally, a pdBF pf is a function from $\mathbb{B}^k$ to $\mathbb{B} \cup \{\star\}$, where $\star$ stands for undefined (cf. [15]). It is equivalently characterized by two sets [7]: its true points $T(pf) = \{\vec{x} \mid pf(\vec{x}) = 1\}$ and its false points $F(pf) = \{\vec{x} \mid pf(\vec{x}) = 0\}$. We denote by $U(pf) = \{\vec{x} \mid pf(\vec{x}) = \star\}$ the *unknown points* of pf. A BF f is an *extension* of a pdBF pf, formally $pf \leq f$, iff $T(pf) \subseteq T(f)$ and $F(pf) \subseteq F(f)$.

We connect partial knowledge of context semantics and pdBFs as follows.

Definition 7. *A pdBF $pf : \mathbb{B}^k \rightarrow \mathbb{B} \cup \{\star\}$ is compatible with a context C_i in an MCS M iff $pf \leq f^{C_i}$ (where f^{C_i} is as in Definition 4).*

Therefore, if a pdBF is compatible with a context, one extension of this pdBF is exactly f^{C_i}, which corresponds to the context's exact semantics.

Example 6 (ctd). Partial knowledge as given in Example 1 can be expressed by the pdBF $pf_{\mathbf{cc}}$ with $T(pf_{\mathbf{cc}}) = \{(0,0,0,0)\}$ and $F(pf_{\mathbf{cc}}) = \{(0,0,A,B) \mid A,B \in \mathbb{B}, (A,B) \neq (0,0)\}$. (See Example 4 for the variable ordering.) □

In the following, a *partially known MCS* (M, i, pf) consists of an MCS M, where context C_i is partially known, given by pdBF pf which is compatible with C_i.

4 Approximations

In this section, we develop a method for calculating under- and overapproximations of diagnoses and explanations, using the pdBF representation for a partially known context C_i. For simplicity, we only consider the case that a single context in the system is partially known (the generalization is straightforward).

Diagnoses. Each diagnosis is defined in terms of consistency, which is witnessed by an output-projected equilibrium. Such an equilibrium requires a certain set of output beliefs O to be accepted by the context C_i, in the presence of certain bridge rule heads I. This means that f_{C_i} has true point $(\vec{\mathrm{I}}, \vec{\mathrm{o}})$. For existence of an equilibrium where C_i gets I as input and accepts O, no more information is required from f_{C_i} than this single true point. We thus can approximate the set of diagnoses of M as follows:

- Completing pf with false points, we obtain the extension $\underline{pf}$ with $T(\underline{pf}) = T(pf)$. The set of diagnoses witnessed by $T(pf)$ contains a *subset* of the diagnoses which actually occur in M, therefore we obtain an *under*approximation.
- Completing pf with true points, we obtain the extension $\overline{pf}$ as the extension of pf with the largest set of true points. The set of diagnoses witnessed by $\overline{pf}$ contains a *superset* of the diagnoses which actually occur in M, providing an *over*approximation. Formally,

Theorem 2. *Given a partially known MCS (M, i, pf), the following holds:*

$$D^{\pm}(M[i/\underline{\mathit{pf}}]) \subseteq D^{\pm}(M) \subseteq D^{\pm}(M[i/\overline{\mathit{pf}}]).$$

Example 7 (ctd). The extensions $\overline{\mathit{pf}}_{\mathbf{cc}}$ and $\underline{\mathit{pf}}_{\mathbf{cc}}$ are as follows:

$$\begin{array}{ll} T(\overline{\mathit{pf}}_{\mathbf{cc}}) = \mathbb{B}^4 \setminus F(\mathit{pf}_{\mathbf{cc}}), & F(\overline{\mathit{pf}}_{\mathbf{cc}}) = F(\mathit{pf}_{\mathbf{cc}}), \\ T(\underline{\mathit{pf}}_{\mathbf{cc}}) = T(\mathit{pf}_{\mathbf{cc}}), \text{ and} & F(\underline{\mathit{pf}}_{\mathbf{cc}}) = \mathbb{B}^4 \setminus T(\mathit{pf}_{\mathbf{cc}}). \end{array}$$

The underapproximation $D^{\pm}(M_2[\mathbf{cc}/\underline{\mathit{pf}}_{\mathbf{cc}}])$ yields several diagnoses, for instance, $D_\alpha = (\{r_{1,moe}\}, \emptyset)$, $D_\beta = (\{r_{2,moe,cnr2}\}, \emptyset)$, and $D_\gamma = (\emptyset, \{r_{3,cnr2}\})$.

The overapproximation $D^{\pm}(M_2[\mathbf{cc}/\overline{\mathit{pf}}_{\mathbf{cc}}])$ contains the empty diagnosis, i.e., $D_\delta = (\emptyset, \emptyset)$, because $M_2[\mathbf{cc}/\overline{\mathit{pf}}_{\mathbf{cc}}]$ is consistent; the latter has the following two equilibria: $(\{\mathit{person}(\mathit{moe})\}, \emptyset)$ and $(\{\mathit{person}(\mathit{moe})\}, \{\mathit{valid}(\mathit{cnr1})\})$. □

Subset-minimality. If we approximate ⊆-minimal diagnoses, the situation is different. Obtaining additional diagnoses may cause an approximated diagnosis to be subset-minimal which is no diagnosis under full knowledge. However, at least one minimal diagnosis under full knowledge is a superset of the former.Vice versa, missing certain diagnoses can yield an approximated subset-minimal diagnoses which is a superset of (at least one) minimal diagnosis. However, if a diagnoses is subset-minimal under both, over- and underapproximation, then it is also a minimal diagnosis under full knowledge.

Theorem 3. *Given a partially known MCS (M, i, pf), the following hold:*

$$\forall D \in D^{\pm}_m(M[i/\underline{\mathit{pf}}])\ \exists D' \in D^{\pm}_m(M) : D' \subseteq D \tag{2}$$

$$\forall D \in D^{\pm}_m(M)\ \exists D' \in D^{\pm}_m(M[i/\overline{\mathit{pf}}]) : D' \subseteq D \tag{3}$$

$$D^{\pm}_m(M[i/\underline{\mathit{pf}}]) \cap D^{\pm}_m(M[i/\overline{\mathit{pf}}]) \subseteq D^{\pm}_m(M) \tag{4}$$

Example 8 (ctd). Note that the diagnoses in Example 7 are in fact the ⊆-minimal diagnoses of the under- and overapproximation, and they are actual ⊆-minimal diagnoses. Under complete knowledge (Example 3), additional ⊆-diagnoses exist which are not obtained by underapproximation. Overapproximation, on the other hand, yields consistency and therefore an empty ⊆-minimal diagnosis D_δ. In Section 5 we develop a strategy for improving this approximation if limited querying of the context is possible. □

We can use the overapproximation to reason about the necessity of bridge rules in actual diagnoses: a necessary bridge rule is present in all diagnoses[2].

Definition 8. *For a set of diagnoses $\mathcal{D}$, the set of* necessary bridge rules *is* $\mathit{nec}(\mathcal{D}) = \{r \mid \forall (D_1, D_2) \in \mathcal{D} : r \in D_1 \cup D_2\}$.

Proposition 1. *Given a partially known MCS (M, i, pf), the set of necessary bridge rules for the overapproximation is necessary in the actual set of diagnoses. This is true for both arbitrary and ⊆-minimal diagnoses:*

$$\mathit{nec}(D^{\pm}(M[i/\overline{\mathit{pf}}])) \subseteq \mathit{nec}(D^{\pm}(M)), \text{ and } \mathit{nec}(D^{\pm}_m(M[i/\overline{\mathit{pf}}])) \subseteq \mathit{nec}(D^{\pm}_m(M)).$$

[2] Note that we do not consider the dual notion of relevance, as it is trivial in our definition of diagnosis: all bridge rules are relevant in any $D^{\pm}(M)$.

While simple, this property is useful in practice: in a repair of an MCS according to a diagnosis, necessary bridge rules need to be fixed in any case.

Inconsistency explanations. So far we have only described approximations for *diagnoses*. We now extend our notions to *inconsistency explanations* (in short 'explanations'), which are dual characterizations to diagnoses [8]. Intuitively, they point out bridge rules such that in the presence of bridge rules E_1 and the absence of bridge rules E_2 the MCS necessarily is inconsistent. Thus explanations allow to separate independent sources of inconsistency, while diagnoses characterize repairs. We first recall their definition.

Definition 9. *Given an MCS M, an* inconsistency explanation *of M is a pair (E_1, E_2) s.t. for all (R_1, R_2) where $E_1 \subseteq R_1 \subseteq br_M$ and $R_2 \subseteq br_M \setminus E_2$, it holds that $M[R_1 \cup heads(R_2)] \models \bot$. By $E^{\pm}(M)$ we denote the set of all inconsistency explanations of M, and by $E^{\pm}_m(M)$ the set of all pointwise subset-minimal ones.*

Example 9. With complete knowledge as in Example 3, there is one $\subseteq$-minimal explanation: $(\{r_{1,moe}, r_{2,moe,cnr2}, r_{3,cnr2}\}, \{r_{2,moe,cnr2}, r_{3,cnr2}\})$. □

Explanations are defined in terms of non-existing equilibria, therefore we can use witnessing equilibria as counterexamples. From the definitions we get:

Proposition 2. *For a given MCS M and a pair $(D_1, D_2) \subseteq br_M \times br_M$ of sets of bridge rules, the following statements are equivalent:*

(i) (D_1, D_2) *is a diagnosis, i.e.,* $(D_1, D_2) \in D^{\pm}(M)$,
(ii) $M[br_M \setminus D_1 \cup heads(D_2)]$ *has an equilibrium, and*
(iii) $(R_1, R_2) = (br_M \setminus D_1, D_2)$ *is a counterexample for all explanation candidates* $(E_1, E_2) \subseteq (br_M \setminus D_1, br_M \setminus D_2)$.

Furthermore, such pairs (D_1, D_2) characterize all counterexamples *that can exist for explanation candidates.*

As a consequence, it is possible to characterize explanations in terms of diagnoses.

Lemma 2. *Given an MCS M, a pair (E_1, E_2) with $E_1, E_2 \subseteq br_M$ is an inconsistency explanation of M iff there exists no diagnosis $(D_1, D_2) \in D^{\pm}(M)$ such that $(D_1, D_2) \subseteq (br_M \setminus E_1, br_M \setminus E_2)$.*

In fact we can sharpen the above by replacing $D^{\pm}$ with $D^{\pm}_m$.

Using this characterization, we can infer the following: a subset of the actual set of diagnoses characterizes a superset of the actual set of explanations. This is true since a subset of diagnoses will rule out a subset of explanations, allowing more candidates to become explanations. Conversely, a superset of diagnoses characterizes a subset of the explanations. Applying Theorem 2, we obtain:

Theorem 4. *Given a partially known MCS (M, i, pf), the following hold:*

$$E^{\pm}(M[i/\overline{pf}]) \subseteq E^{\pm}(M) \subseteq E^{\pm}(M[i/\underline{pf}])$$
$$\forall E \in E^{\pm}_m(M[i/\overline{pf}])\ \exists E' \in E^{\pm}_m(M) : E' \subseteq E$$
$$\forall E \in E^{\pm}_m(M)\ \exists E' \in E^{\pm}_m(M[i/\underline{pf}]) : E' \subseteq E$$

Therefore, the extensions $\overline{pf}$ and $\underline{pf}$ allow to underapproximate and overapproximate diagnoses as well as inconsistency explanations.

Example 10 (ctd). From $\underline{pf}_{\mathbf{cc}}$ as in Example 7 we obtain one $\subseteq$-minimal explanation: $E_\mu = (\{r_{1,moe}, r_{2,moe,cnr2}\}, \{r_{3,cnr2}\})$. This explanation is a subset of the actual minimal explanation in Example 9. □

5 Limited Querying

Up to now we used existing partial knowledge to approximate diagnoses, assuming that more information is simply not available. However, in practical scenarios like our running example, one can imagine that a (small) limited number of queries to a partially known context can be issued. Therefore we next aim at identifying queries to contexts, such that incorporating their answers into the pdBF will yield the best guarantee of improvement in approximation accuracy.

Given a partially known MCS (M, i, pf), let $D^{\pm}_{\Delta}(M, i, pf) = D^{\pm}(M[i/\overline{pf}]) \setminus D^{\pm}(M[i/\underline{pf}])$ (in short: $D^{\pm}_{\Delta}(pf)$ or $D^{\pm}_{\Delta}$) be the set of *potential diagnoses*, which are possible from the overapproximation but unconfirmed by the underapproximation. A large set of potential diagnoses provides less information than a smaller set. Hence, we aim at identifying unknown points of pf which remove from $D^{\pm}_{\Delta}$ as many potential diagnoses as possible. To this end we introduce the concept of a *witness* as an unknown point and a potential diagnosis that is supported by this point if it is a true point.

Definition 10. *Given a partially known MCS (M, i, pf), a* witness *is a pair $(\vec{\mathrm{X}}, D)$ s.t. $\vec{\mathrm{X}} \in U(pf)$ and $D \in D^{\pm}(M[i/f_{\vec{\mathrm{X}}}]) \cap D^{\pm}_{\Delta}$, where $f_{\vec{\mathrm{X}}}$ is the BF with the single true point $T(f_{\vec{\mathrm{X}}}) = \{\vec{\mathrm{X}}\}$. We denote by $W_{(M,i,pf)}$ the set of all witnesses wrt. (M, i, pf). If clear from the context, we omit subscript (M, i, pf).*

Based on W we define the set $wnd(\vec{\mathrm{X}}) = \{D \mid (\vec{\mathrm{X}}, D) \in W\}$ of potential diagnoses witnessed by unknown point $\vec{\mathrm{X}}$, and the set $ewnd(\vec{\mathrm{X}}) = \{D \in wnd(\vec{\mathrm{X}}) \mid \nexists \vec{\mathrm{X}}' \neq \vec{\mathrm{X}} : (\vec{\mathrm{X}}', D) \in W\}$ of potential diagnoses exclusively witnessed by $\vec{\mathrm{X}}$. These sets are used to investigate how much the set of potential diagnoses is reduced when adding information about the value of an unknown point $\vec{\mathrm{X}}$ to pf.

Lemma 3. *Given a partially known MCS (M, i, pf), and $\vec{\mathrm{X}} \in U(pf)$, let $pf_{\vec{\mathrm{X}}:0}$ ($pf_{\vec{\mathrm{X}}:1}$) the pdBF that results from pf by making $\vec{\mathrm{X}}$ a false (true) point. Then $D^{\pm}_{\Delta}(pf_{\vec{\mathrm{X}}:1}) = D^{\pm}_{\Delta}(pf) \setminus wnd(\vec{\mathrm{X}})$, and $D^{\pm}_{\Delta}(pf_{\vec{\mathrm{X}}:0}) = D^{\pm}_{\Delta}(pf) \setminus ewnd(\vec{\mathrm{X}})$.*

Note that $ewnd(\vec{\mathrm{X}}) \subseteq wnd(\vec{\mathrm{X}}) \subseteq D^{\pm}_{\Delta}$. If $\vec{\mathrm{X}}$ is a true point, $|wnd(\vec{\mathrm{X}})|$ many potential diagnoses become part of the underapproximation; otherwise $|ewnd(\vec{\mathrm{X}})|$ many potential diagnoses are no longer part of the overapproximation. Knowing the value of $\vec{\mathrm{X}}$ therefore guarantees a reduction of $D^{\pm}_{\Delta}$ by $|ewnd(\vec{\mathrm{X}})|$ diagnoses.

Proposition 3. *Given a partially known MCS (M, i, pf), for all $\vec{\mathrm{X}} \in U(pf)$ such that the cardinality of $ewnd(\vec{\mathrm{X}})$ is maximal, the following holds:*

$$\max_{u \in \mathbb{B}} \left| D^{\pm}_{\Delta}\left(pf_{\vec{\mathrm{X}}:u}\right) \right| \leq \min_{\vec{\mathrm{Y}} \in U(pf)} \max_{v \in \mathbb{B}} \left| D^{\pm}_{\Delta}\left(pf_{\vec{\mathrm{Y}}:v}\right) \right| . \tag{5}$$

Proposition 3 suggests to query unknown points $\vec{x}$ where $|ewnd(\vec{x})|$ is maximum. If there are more false points than true points (e.g., for contexts that accept only one belief set for each input), using $ewnd$ instead of wnd is even more suggestive. If the primary interest are necessary bridge rules (cf. previous section), we can base query selection on the number of bridge rules which become necessary if a certain unknown point is a false point. Let $nwnd(\vec{x}) = nec(\overline{D^{\pm}} \setminus ewnd(\vec{x})) \setminus nec(\overline{D^{\pm}})$, where $\overline{D^{\pm}} = D^{\pm}(M[i/\overline{pf}])$, then $|nwnd(\vec{x})|$ many bridge rules become necessary if $\vec{x}$ is identified as a false point.

Another possible criterion for selecting queries can be based on the likelihood of errors, similar to the idea of *leading diagnoses* [10]. Although a different notion of diagnosis is used there, the basic idea is applicable to our setting: if multiple problematic bridge rules are less likely than single ones, or if we have confidence values for bridge rules (e.g., some were designed by an expert, others by a less experienced administrator), then we can focus confirming or discarding diagnoses that have a high probability. If we have equal confidence in all bridge rules, this amounts to using *cardinality-minimal* potential diagnoses for determining witnesses and guiding the selection of queries.

Example 11 (ctd). In our example, the set of potential diagnoses is large, but the cardinality-minimal diagnosis is the empty diagnosis, which has the following property: bridge rule input at $C_{\mathbf{cc}}$ is $\{card(cnr2)\}$, and $C_{\mathbf{cc}}$ either accepts $\emptyset$ or $\{valid(cnr1)\}$ (the unrelated credit card). Therefore, points $(0, 1, 0, 0)$ and $(0, 1, 1, 0)$ are the only witnesses for D_δ, and querying these two unknown points is sufficient for verifying or falsifying D_δ. (Note that $pf_{\mathbf{cc}}$ has 12 unknown points, the four known points (one true and three false points) are $(0, 0, X, Y)$ s.t. $X, Y \in \mathbb{B}$.) After updating pf with these points (false points, if all credit cards are valid), the overapproximation yields the $\subseteq$-minimal diagnoses; this result is optimal. $\square$

Instead of membership queries which check whether $O \in \mathbf{ACC}(kb \cup I)$ for given $(\vec{\imath}, \vec{o})$, one could use stronger queries that provide the *value* of $\mathbf{ACC}(kb \cup I)$ for a given $\vec{\imath}$. On the one hand this allows for a better query selection, roughly speaking because combinations of unknown points witness more diagnoses exclusively than they do individually. On the other hand, considering such combinations increases computational cost. Another extension of limited querying is the usage of meta-information, e.g., monotonicity, or consistency properties, of a partially known context.

6 Discussion

Approximation Quality. In the previous section, we related unknown points to potential diagnoses. This correspondence allows to obtain an estimate for the quality of an approximation, simply by calculating the ratio between known and potential true (resp., false) points: a high value of $\frac{|T(pf)|}{|T(pf)|+|U(pf)|}$ indicates a high underapproximation quality, while a low value indicates an underapproximation distant from the actual system. This is analogous for overapproximation, exchanging $T(pf)$ with $F(pf)$. These estimates can be calculated efficiently and prior to calculating an approximation; a decision between under- and overapproximation could be based on this heuristic. Concerning quality note also that even if nothing is known about the behavior of some context C, the overapproximation accurately characterizes inconsistencies that do not involve C.

Complexity and Computation. Since our approximation methods deal with incomplete knowledge, it is important how their computational complexity compares to the full knowledge case. For the latter setting, the following results were established in [8], depending on the complexity of output checking for contexts C_i, which is deciding for $C_i, I \subseteq IN_i$ and $O \subseteq OUT_i$ whether $O \in \mathbf{ACC}(kb_i \cup I)|_{OUT_i}$. With output checking in $\mathbf{P}$ (resp., $\mathbf{NP}$, $\mathbf{\Sigma_k^P}$), recognizing correct diagnoses is in $\mathbf{NP}$ (resp., $\mathbf{NP}$, $\mathbf{\Sigma_k^P}$) while recognizing minimal diagnoses and minimal explanations is in $\mathbf{D^P}$ (resp., $\mathbf{D^P}$, $\mathbf{D_k^P}$); completeness holds in all cases.

Let us first consider the case where some contexts C_i are given by their corresponding BF f_i (such that $f_i(\vec{\mathrm{I}}, \vec{\mathrm{o}})$ can be evaluated efficiently). As we know that context C_i accepts only input/output combinations which are true points of f, we simply guess all possible output beliefs O_i of all contexts and evaluate bridge rules to obtain I_i; if for some C_i as above, $f_i(\vec{\mathrm{I}}_i, \vec{\mathrm{o}}_i)$=0 we reject, otherwise we continue checking context acceptance for other contexts. Overall, this leads to the same complexity as if all contexts were total. Thus, detecting explanations of inconsistency for an MCS M, where some contexts are given as BFs, has the same complexity as if M were given regularly.

Approximations are done on an MCS where a pdBF pf is given instead of a BF f, in a representation such that the value of $pf(\vec{\mathrm{I}}, \vec{\mathrm{o}})$ can be computed efficiently. This implies that the extensions $\underline{pf}$ and $\overline{pf}$ can be computed efficiently as well. Hence, approximations of diagnoses and explanations have the same complexity as the exact concepts. Dealing with incomplete information usually increases complexity, as customary for many nonmonotonic reasoning methods. Our approach, however, exhibits no such increase in complexity, even though it provides faithful under- and overapproximations.

Learning. To learn a BF seems suggestive for our setting of incomplete information. However, explaining inconsistency requires correct information, therefore pac-learning methods [15] are not applicable. On the other hand, exact methods [1] require properties of the contexts which are beneficial to learning and might not be present[3]. Furthermore, contexts may only allow membership queries, which are insufficient for efficient learning of many concept domains [1]. Furthermore, partially known contexts may not allow many, even less a polynomial number of queries (which is the target for learnability). Most likely it will thus not be possible to learn the complete function. Hence learning cannot replace our approach, but it can be useful as a preprocessing step to increase the amount of partial information.

7 Related Work and Conclusion

To the best of our knowledge, explaining inconsistency in multi-context systems with partial specification has not been addressed before. Weakly related to our work is [14], who aimed at approximating abductive diagnoses of a single knowledge base. They replaced classical entailment with approximate entailment of [12], motivated by

[3] Note that, even if a context's logic is monotonic (resp., positive) this does not imply that the BF corresponding to the context is monotonic (resp., positive).

computational efficiency. However, there is no lack of information about the knowledge base or semantics as in our case.

Our over- and underapproximations of $D^{\pm}$ and $E^{\pm}$ are reminiscent of lower and upper bounds of classical theories (viewed as sets of models [13]), known as cores and envelopes. The latter also were used for (fast) sound, resp. complete, reasoning from classical theories.

The limited querying approach is related to optimal probing strategies [6]. However, we do not require probing to localize faults in the system, but to obtain information about the behavior of system parts, which have a much more fine grained inner structure and more intricate dependencies than the systems in [6]. (Those system parts have as possible states ‘up’, and ‘down’, while in MCSs each partially known context possibly accepts certain belief sets for certain inputs.)

Ongoing further work includes an implementation of the approach given in this paper, and the usage of metainformation about context properties to improve approximation accuracy. The incorporation of probabilistic information into the pdBF representation is another interesting topic for future research.

References

1. Angluin, D.: Queries and concept learning. Machine Learning 2, 319–342 (1988)
2. Bikakis, A., Antoniou, G.: Distributed defeasible contextual reasoning in ambient computing. In: Aarts, E., Crowley, J.L., de Ruyter, B., Gerhäuser, H., Pflaum, A., Schmidt, J., Wichert, R. (eds.) AML 2008. LNCS, vol. 5355, pp. 308–325. Springer, Heidelberg (2008)
3. Bonatti, P.A., Olmedilla, D.: Rule-based policy representation and reasoning for the semantic web. In: Antoniou, G., Aßmann, U., Baroglio, C., Decker, S., Henze, N., Patranjan, P.-L., Tolksdorf, R. (eds.) Reasoning Web. LNCS, vol. 4636, pp. 240–268. Springer, Heidelberg (2007)
4. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI, pp. 385–390 (2007)
5. Brewka, G., Roelofsen, F., Serafini, L.: Contextual default reasoning. In: IJCAI, pp. 268–273 (2007)
6. Brodie, M., Rish, I., Ma, S., Odintsova, N.: Active probing strategies for problem diagnosis in distributed systems. In: IJCAI, pp. 1337–1338 (2003)
7. Crama, Y., Hammer, P.L., Ibaraki, T.: Cause-effect relationships and partially defined boolean functions. Annals of Operations Research 16, 299–326 (1988)
8. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in nonmonotonic multi-context systems. In: KR, pp. 329–339 (2010)
9. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics, or: How we can do without modal logics. Artificial Intelligence 65(1), 29–70 (1994)
10. de Kleer, J.: Focusing on probable diagnoses. In: AAAI, pp. 842–848 (1991)
11. McCarthy, J.: Notes on formalizing context. In: IJCAI, pp. 555–562 (1993)
12. Schaerf, M., Cadoli, M.: Tractable reasoning via approximation. Artificial Intelligence 74(2), 249–310 (1995)
13. Selman, B., Kautz, H.: Knowledge Compilation and Theory Approximation. J. ACM 43(2), 193–224 (1996)
14. ten Teije, A., van Harmelen, F.: Computing approximate diagnoses by using approximate entailment. In: KR, pp. 256–265 (1996)
15. Valiant, L.G.: A theory of the learnable. Commun. ACM 27, 1134–1142 (1984)

Relational Information Exchange and Aggregation in Multi-Context Systems*

Michael Fink, Lucantonio Ghionna, and Antonius Weinzierl

Institute of Information Systems
Vienna University of Technology
Favoritenstraße 9-11, A-1040 Vienna, Austria
{fink,weinzierl}@kr.tuwien.ac.at, l.ghionna@mat.unical.it

Abstract. Multi-Context Systems (MCSs) are a powerful framework for representing the information exchange between heterogeneous (possibly nonmonotonic) knowledge-bases. Significant recent advancements include implementations for realizing MCSs, e.g., by a distributed evaluation algorithm and corresponding optimizations. However, certain enhanced modeling concepts like aggregates and the use of variables in bridge rules, which allow for more succinct representations and ease system design, have been disregarded so far.

We fill this gap introducing open bridge rules with variables and aggregate expressions, extending the semantics of MCSs correspondingly. The semantic treatment of aggregates allows for alternative definitions when so-called grounded equilibria of an MCS are considered. We discuss options in relation to well-known aggregate semantics in answer-set programming. Moreover, we develop an implementation by elaborating on the DMCS algorithm, and report initial experimental results.

1 Introduction

The Multi-Context System (MCS) formalism is a flexible and powerful tool to realize the information exchange between heterogeneous knowledge bases. An MCS captures the information available in a number of contexts, each consisting of a knowledge base, represented in a given 'logic', e.g., classical logic, description logics or logic programs under answer set semantics, and a set of so-called *bridge rules* modeling the information exchange between contexts.

Initial developments of the formalism [11] have been complemented with relevant language extensions, e.g., to incorporate nonmonotonic reasoning [4] or preferences [1], and recently algorithms and implementations have been devised to compute (partial) equilibria, i.e. the semantics of MCSs, in a distributed setting [2]. Through these research efforts the formal framework has become amenable for practical application.

However, most KR formalisms that have successfully been applied in real world scenarios build on a predicate logic setting, rather than on a propositional language, since the former allows for a more succinct representation. This eases the modeling task for the knowledge engineer and often is essential to make the representation task practically feasible. MCSs are flexible enough to incorporate such formalisms, but for

* Supported by the Vienna Science and Technology Fund (WWTF) under grant ICT08-020.

modeling the information exchange one must specify the intended knowledge exchange by 'ground' bridge rules, because bridge rules currently do not allow for variables.

Moreover, an important aspect in relevant scenarios is the possibility to aggregate data, for instance in case of accounting to sum up sales. For standard database query languages, dedicated language constructs exist for aggregation and more recently, aggregates have become a hot topic in Answer Set Programming (ASP), e.g., cf. [7,14,8,13,12]. However, MCSs for such scenarios may include contexts that merely represent relational data, like RDF stores, where the logic just provides a simple means for querying that is not capable to express aggregation. Allowing for aggregate constructs in bridge rules of an MCS, enables aggregation even if the logic of a context lacks this expressivity. Additionally, it goes beyond aggregation within a particular context formalism since it enables a form of aggregation, where data is collected from different contexts.

Example 1. Consider a company selling various products. The logistics department C_1 has to refit a shop C_2, if more than 25 products of a certain model category have been sold. Sales management C_3 is responsible for product categorization and placement. The quantity to be delivered for a product depends on the free storage capacity at the shop for that product, and on whether it is on sale, as decided by sales management. We would like to represent the respective information exchange in an MCS with bridge rules like:

$$(1{:}\,\mathit{refit}) \leftarrow\ SUM\{Y, X : (2{:}\,\mathit{sold}(X,Y)), (3{:}\,\mathit{cat}(X,\mathit{mc}))\} \geq 25. \tag{1}$$

$$(1{:}\,\mathit{deliv}(X,Y)) \leftarrow\ (2{:}\,\mathit{cap}(X,Y)), \mathit{not}(3{:}\,\mathit{off\_sale}(X)). \tag{2}$$

Aggregating information from different knowledge sources is an important task in various application domains, e.g., in bioinformatics or linguistics, where for instance the number of occurrences of specific words in a text corpus is combined with grammatical relations between them. To cope with these in current MCSs one might introduce additional contexts with suitable logics for the aggregation. However, such an approach incurs a significant overhead in representing the information exchange.

The research issues addressed in this work thus are: first to enable a more succinct representation of the information exchange in MCSs, and second to extend their modeling capabilities wrt. information aggregation. This is achieved by introducing so-called *open bridge rules* that may contain variables and aggregate expressions. Our contributions are summarized as follows:

– We formalize above intuitions, defining syntax and semantics—in terms of equilibria—of so-called *relational MCSs with aggregates*, lifting the framework in [3] and extending bridge rules with aggregate atoms based on basic notions in [7]. We show that this lifting causes an exponential increase of complexity.
– We study semantic alternatives, in particular *grounded equilibria* that avoid self-justification of beliefs via bridge rules incorporating foundedness criteria. By the introduction of aggregates, due to their potential nonmontonicity, different semantic options exist. We provide several definitions and correspondence results in analogy to well-known answer set semantics in presence of aggregates [7,14,8].
– Extending a recently developed algorithm and its implementation [2] in order to handle relational MCSs and aggregates, we develop an implementation, called

DMCSAgg, for the distributed computation of (partial) equilibria. Respective initial experimental results are reported and discussed briefly.

Overcoming current shortcomings of MCSs concerning the treatment of relational information and aggregation in theory and implementation, our work pushes the MCS framework further towards practical applicability.

2 Preliminaries

Multi-Context Systems as defined in [3] build on an abstract notion of a *logic* L as a triple (KB_L, BS_L, ACC_L), where KB_L is the set of well-formed knowledge bases of L, BS_L is the set of possible belief sets, and $ACC_L : KB_L \rightarrow 2^{BS_L}$ is a function describing the semantics of L by assigning each knowledge-base a set of acceptable sets of beliefs.

A *Multi-Context System* $M = (C_1, \ldots, C_n)$ is a collection of contexts $C_i = (L_i, kb_i, br_i)$ where L_i is a logic, $kb_i \in KB_{L_i}$ a knowledge base and br_i a set of bridge rules of the form:

$$(k\colon s) \leftarrow (c_1\colon p_1), \ldots, (c_j\colon p_j), not(c_{j+1}\colon p_{j+1}), \ldots, not(c_m\colon p_m), \tag{3}$$

such that $1 \leq k \leq n$ and $kb \cup s$ is an element of KB_{L_k}, as well as $1 \leq c_\ell \leq n$ and p_ℓ is element of some belief set of BS_{c_ℓ}, for all $1 \leq \ell \leq m$. For a bridge rule r, by $hd(r)$ we denote the belief s in the head of r, while $body(r)$, $pos(r)$, and $neg(r)$ denote the sets $\{(c_{\ell_1}\colon p_{\ell_1}), not(c_{\ell_2}\colon p_{\ell_2})\}$, $\{(c_{\ell_1}\colon p_{\ell_1})\}$, and $\{(c_{\ell_2}\colon p_{\ell_2})\}$, respectively, where $1 \leq \ell_1 \leq j$ and $j < \ell_2 \leq m$.

A belief state $S = (S_1, \ldots, S_n)$ is a belief set for every context, i.e., $S_i \in BS_i$. A bridge rule r of form (3) is applicable wrt. S, denoted by $S \models body(r)$, iff $p_\ell \in S_{c_\ell}$ for $1 \leq \ell \leq j$ and $p_\ell \notin S_{c_\ell}$ for $j \leq \ell \leq m$. We denote the heads of all applicable bridge rules of context C_i wrt. belief state S by $app_i(S) = \{hd(r) \mid r \in br_i \wedge S \models body(r)\}$.

The semantics of an MCS is defined in terms of equilibria where an *equilibrium* $(S_1, \ldots, S_n)$ is a belief state such that $S_i \in ACC_i(kb_i \cup app_i(S))$.

Aggregates in Answer-Set Programming provide a suitable basis for the introduction of aggregate atoms in bridge rules. In particular we introduce aggregates for MCSs adopting the syntax and elementary semantic notions of [7]. We assume familiarity with non-ground ASP syntax and semantics and just briefly recall relevant concepts.

A standard literal is either an atom a or a default negated atom $not\ a$. A *set term* is either a *symbolic set* of the form $\{Vars\colon Conj\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of standard literals, or a *ground set*, i.e., set of pairs $\langle \bar{t}\colon Conj\rangle$, where $\bar{t}$ is a list of constants and $Conj$ is a conjunction of ground atoms. If f is an aggregate function symbol and S is a set term, then $f(S)$ is called an *aggregate function*, mapping a multiset of constants to an integer. An *aggregate atom* is an expression of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{=, <, \leq, >, \geq\}$, and T is a variable or a constant called guard. A (normal)[1] logic program with aggregates, a program for short, is a set of rules $a \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_m$. where a is an

[1] Anticipating the following adaption to MCSs, we disregard disjunction in rule heads.

atom and b_j is either an atom or an aggregate atom, for all $1 \leq j \leq m$. A rule is safe iff (a) every variable in a standard atom of r (i.e., a global variable of r) also appears in a positive, i.e., not default negated, standard literal in the body of r, (b) all other variables (local variables) of a symbolic set also appear in a positive literal in $Conj$, and (c) each guard is either a constant or a global variable of r. A program is safe iff all its rules are safe, and for the definition of semantics we restrict to safe programs.

A *ground instance* of a rule r of P is obtained in a two-step process using the Herbrand universe U_P of P: first a substitution is applied to the set of global variables of r, after that every symbolic set S is replaced by its instantiation, which is defined as the ground set consisting of all pairs obtained by applying a substitution to the local variables of S. The *grounding* of a program P is the set of all ground instances of rules in P. An *interpretation* is a set $I \subseteq B_P$, where B_P denotes the standard Herbrand base of a program P, i.e., the set of standard atoms constructible from (standard) predicates and constants of P. For (a conjunction of) standard literals F, $I \models F$ is defined as usual. Let S be a ground set, the valuation of S wrt. I is the multiset $I(S) = [t_1 \mid \langle t_1, \ldots, t_n{:}Conj\rangle \in S \wedge I \models Conj]$. The valuation $I(f(S))$ of a ground aggregate function $f(S)$ is the result of applying f on $I(S)$, and void if $I(S)$ is not in the domain of f. Then $I \models a$ for a ground aggregate atom $a = f(S) \prec k$ iff $I(f(S))$ is not void and $I(f(S)) \prec k$ holds. Models of ground rules and programs are defined as usual. Given this setting, different semantics can be given to ground programs. A discussion follows in Section 4 and we refer to [8,7,14] for further details.

3 MCSs with Relational Beliefs and Aggregates

Our goal is to extend the modeling capabilities of the MCS framework in two ways. First, we aim at a more succinct representation of the information exchange via bridge rules in cases where we have additional information on the structure of the accessed beliefs. Second, we introduce a means to aggregate information from different contexts.

3.1 Relational Multi-Context Systems

In order to treat certain beliefs, respectively elements of a knowledge base, as relational, we first slightly generalize the notion of a logic, allowing to explictly declare certain beliefs and elements of a knowledge base to be relational.

A *relational* logic L is a quadruple $(KB_L, BS_L, ACC_L, \Sigma_L)$, where KB_L, BS_L, and ACC_L are as before, and Σ_L is a signature consisting of sets P_L^{KB} and P_L^{BS} of predicate names p with an associated arity $ar(p) \geq 0$, and a universe U_L, i.e., a set of object constants, such that $(P_L^{KB} \cup P_L^{BS}) \cap U_L = \emptyset$. Let $B_L^{\chi} = \{p(c_1, \ldots, c_k) \mid p \in P_L^{\chi} \wedge ar(p) = k \wedge \forall 1 \leq i \leq k : c_i \in U_L\}$, for $\chi \in \{KB, BS\}$. An element of B_L^{χ} is termed *relational* and *ground*, and it has to be a an element of some knowledge base in KB_L if $\chi{=}KB$, otherwise, if $\chi{=}BS$, it is required to be an element of some belief set in BS_L. A knowledge base element not in B_L^{KB}, respectively a belief not in B_L^{BS} is called *ordinary*. Note that every logic $L = (KB_L, BS_L, ACC_L)$ in the original sense, can be regarded as a relational logic with an empty signature, i.e., where $P_L^{KB}{=}P_L^{BS}{=}U_L{=}\emptyset$.

Example 2. For instance, consider non-ground normal logic programs (NLPs) under ASP semantics over a signature $\mathcal{L}$ of pairwise disjoint sets of predicate symbols Pred, variable symbols Var, and constants Cons. Then, KB is the set of non-ground NLPs over $\mathcal{L}$, BS is the power set of the set of ground atoms over $\mathcal{L}$, $ACC(kb)$ is the set of kb's answer sets, $P^{KB}{=}P^{BS}{=}\mathit{Pred}$, and $U{=}\mathit{Cons}$.

Next we introduce variables in bridge rules. Given a set of relational logics $\{L_1, \ldots, L_n\}$, let V denote a countable set of variable names, s.t. $V \cap \bigcup_{i=1}^{n}(P^{KB}_{L_i} \cup P^{BS}_{L_i} \cup U_{L_i}) = \emptyset$. A (possibly non-ground) relational element of L_i is of the form $p(t_1, \ldots, t_k)$, where $p \in P^{KB}_{L_i} \cup P^{BS}_{L_i}$, $ar(p) = k$, and t_j is a term from $V \cup U_{L_i}$, for $1 \leq j \leq k$. A *relational bridge rule* is of the form (3) such that s is ordinary or a relational knowledge base element of L_k, and $p_1, \ldots, p_m$ are either ordinary or relational beliefs of L_ℓ, $1 \leq \ell \leq m$. A relational MCS consists of contexts with relational logics and bridge rules.

Definition 1 (Relational MCS). *A* relational MCS $M = (C_1, \ldots, C_n)$ *is a collection of contexts* $C_i = (L_i, kb_i, br_i, D_i)$*, where* L_i *is a relational logic,* kb_i *is a knowledge base,* br_i *is a set of relational bridge rules, and* D_i *is a collection of import domains* $D_{i,\ell}$*,* $1 \leq \ell \leq n$*, such that* $D_{i,\ell} \subseteq U_\ell$*.*

By means of import domains, one can explicitly restrict the relational information of context C_j accessed by an open relational bridge rule of context C_i. In the following we restrict to finite import domains, and unless stated otherwise, we assume that import domains $D_{i,\ell}$ are implicitly given by corresponding active domains D^A_ℓ, as follows. The active domain D^A_i of a context C_i is the set of object constants appearing in kb_i or in $hd(r)$ for some $r \in br_i$ such that $hd(r)$ is relational.

Example 3 (Ex. 1 ctd.). Let $M_1 = (C_1, C_2, C_3)$ be an MCS for the company of Ex. 1, where L_1 and L_3 are ASP logics as in Ex. 2, and the shop C_2 uses a relational database with a suitable logic L_2. For simplicity, consider just two products p_1 and p_2, and simplified kb's: $kb_1 = \{\}$, $kb_2 = \{\mathit{cap}(p_1, 11), \mathit{cap}(p_2, 15)\}$, and $kb_3 = \{\mathit{off\_sale}(p_2)\}$. Import domains contain p_1, p_2, and a domain of integers, say $1, \ldots, 25$. Let us further restrict to a single bridge rule r_1, rule (2), in br_1. Then, M_1 is a relational MCS, and $\mathit{deliv}(X, Y)$, $\mathit{cap}(X, Y)$, and $\mathit{off\_sale}(X)$ are relational elements of r_1.

The semantics of a relational MCS is defined in terms of grounding. A *ground instance* of a relational bridge rule $r \in br_i$ is obtained by an admissible substitution of the variables in r to object constants in $\bigcup_{\ell=1}^{m} D_{i,\ell}$. A substitution of a variable X with constant c is admissible iff c is in the intersection of the domains of all occurrences of X in r, where $D_{i,i}$ is the domain for an occurrence in $hd(r)$ and $D_{i,\ell}$ is the domain for an occurrence in some $(c_\ell\colon p_\ell(\overline{t}))$ of $pos(r) \cup neg(r)$. The *grounding* of a relational MCS M, denoted as $grd(M)$ consists of the collection of contexts obtained by replacing br_i with the set $grd(br_i)$ of all ground instances of every $r \in br_i$. The notions of belief state and applicability of a bridge rule wrt. a belief state apply straight forwardly to a relational MCS M and to $grd(M)$, respectively. In slight abuse of notation, let us reuse $app_i(S)$ to denote the set $\{hd(r) \mid r \in grd(br_i) \wedge S \models r\}$, for a belief state S.

Definition 2 (Equilibrium of a Relational MCS). *Given a relational MCS* M*, a belief state* $S = (S_1, \ldots, S_n)$ *of* M *is an equilibrium iff* $S_i \in ACC_i(kb_i \cup app_i(S))$*.*

Unless stated otherwise we consider relational MCSs, simply referred to as MCSs.

Example 4 (Ex. 3 ctd.). The grounding of bridge rule r_1 in the grounding of M_1 is given by the rules $(1\colon \mathit{deliv}(\chi, \kappa)) \leftarrow (2\colon \mathit{cap}(\chi, \kappa)), \mathit{not}(3\colon \mathit{off\_sale}(\chi))$, where $\chi \in \{p_1, p_2\}$ and $\kappa \in \{1, \ldots, 25\}$. The only equilibrium of M_1 is $S = (\{\mathit{deliv}(p_1, 11)\}, S_2, S_3)$, where S_2 and S_3 contain the relations in kb_2 and kb_3, respectively.

3.2 Aggregates

To incorporate a means for information aggregation into the MCS framework, we introduce aggregate atoms in bridge rules, following the approach of [7] (see Sec. 2).

Let $M = (C_1, \ldots, C_n)$ be an MCS, and let V be a corresponding set of variable names. A bridge atom is of the form $(c_i\colon p_i)$, where $1 \leq i \leq n$ and p_i is an ordinary or relational belief. A bridge literal is a bridge atom or a bridge atom preceded by not. A *symbolic set* is of the form $\{\mathit{Vars}\colon \mathit{Conj}\}$, where Vars is a list of variables from V and Conj is a conjunction of bridge literals; a *ground set* is a set of pairs $\langle \bar{t}\colon \mathit{Conj}\rangle$, where $\bar{t}$ is a list of constants from $\bigcup_{i=1}^{n} U_i$ and Conj is a conjunction of positive bridge literals over ordinary and ground relational beliefs. On the basis of these slightly adapted definitions of symbolic and ground sets, we consider set terms, aggregate functions and aggregate atoms as in Section 2.

Definition 3 (Bridge Rule with Aggregates). *Let $M = (C_1, \ldots, C_n)$ be an MCS. A (relational)* bridge rule with aggregates *is of the form*

$$(k\colon s) \leftarrow a_1, \ldots a_{j-1}, \mathit{not}\, a_j, \ldots, \mathit{not}\, a_m, \tag{4}$$

where $1 \leq k \leq n$, s is an ordinary or relational knowledge base element of L_k, and a_i is either a bridge atom, or an aggregate atom.

For a bridge rule with aggregates r of the form (4), let $\mathit{body}(r) = \{a_1, \ldots a_{j-1}, \mathit{not}\, a_j, \ldots, \mathit{not}\, a_m\}$, $\mathit{pos}(r) = \{a_1, \ldots a_{j-1}\}$ and $\mathit{neg}(r) = \{a_j, \ldots a_m\}$. Variables that appear in $\mathit{hd}(r)$ or in a relational belief in $\mathit{pos}(r) \cup \mathit{neg}(r)$ are called *global variables*, variables of r that are not global are *local variables*. A bridge rule with aggregates r is *safe* iff (i) all local variables of a symbolic set in r also appear in a belief literal in Conj, and (ii) each guard is either a constant or a global variable of r.

An MCS with aggregates is an MCS where the set of bridge rules br_i of a context C_i is a set of safe bridge rules with aggregates. In the following we consider safe bridge rules with aggregates and simply refer to them as bridge rules.

Example 5 (Ex. 4 ctd.). Consider the extension M_2 of the MCS M_1, where $br_1 = \{r_1, r_2\}$ and r_2 is the bridge rule (1). The resulting MCS M_2 is a relational MCS with aggregates. Note that r_2 is safe since the guard is a constant and due to $(2 : \mathit{sold}(X, Y))$.

To give the semantics of an MCS with aggregates, the notions of grounding and applicability of ground bridge rules are extended, taking aggregate atoms into account.

The *instantiation of a symbolic set* is the ground set consisting of all pairs obtained by applying a substitution to the local variables of S, which is an admissible substitution (as defined for a relational MCS above) for the local variables in Conj. A *partial ground* *instance* of a bridge rule r of an MCS M with aggregates is obtained by an admissible substitution of the global variables of r. A *ground instance* of r is obtained from a partial ground instance r' replacing every symbolic set S in r' by its instantiation. The *grounding* of an MCS M with aggregates $grd(M)$ consists of the collection of contexts where br_i is replaced with $grd(br_i)$,i.e., the set of all ground instances for every $r \in br_i$.

Given a belief state $S = (S_1, \dots, S_n)$, we say that S satisfies a bridge atom $a = (c_i{:}\, p_i)$, in symbols $S \models a$, iff $p_i \in S_i$. A negative bridge literal $not\, a$ is satisfied if $p_i \notin S_i$, thus $S \models not\, a$ iff $S \not\models a$. For a conjunction of bridge literals F, $S \models F$ is defined as usual. Let A be a ground set, the *valuation of* A *wrt.* S is the multiset $S(A) = [t_1 \mid \langle t_1, \dots, t_n{:}Conj\rangle \in A \wedge S \models Conj]$. The valuation $S(f(A))$ of a ground aggregate function $f(A)$ is the result of applying f on $S(A)$, and void if $S(A)$ is not in the domain of f. Furthermore, $S \models a$ for a ground aggregate atom $a = f(A) \prec k$ iff $S(f(A))$ is not void and $S(f(A)) \prec k$ holds; for a default negated ground aggregate atom $not\, a$, again $S \models not\, a$ iff $S \not\models a$.

Definition 4 (Applicable Bridge Rule). *Given an MCS M with aggregates and a belief state $S = (S_1, \dots, S_n)$, a ground instance $r \in grd(br_i)$ is* applicable wrt. S*, i.e., $S \models body(r)$, iff $S \models a$ for all a in $body(r)$.*

The set of applicable heads of context i, denoted $app_i(S)$, is given by $app_i(S) = \{hd(r) \mid r \in grd(br_i) \wedge S \models body(r)\}$. A belief state $S = (S_1, \dots, S_n)$ is an *equilibrium* of an MCS with aggregates iff $S_i \in ACC_i(kb_i \cup app_i(S))$, for $1 \leq i \leq n$.

Example 6 (Ex. 5 ctd.). Assume the shop has sold 18 and 7 quantities of p_1 and p_2, respectively, and let M_2' be the MCS obtained from M_2, where $kb_1' = kb_1$, $kb_2' = kb_2 \cup \{sold\,(18, p_1)\,, sold\,(7, p_2)\}$, and $kb_3' = kb_3 \cup \{cat\,(p_1, mc)\,, cat\,(p_2, mc)\}$. Let $\phi(\kappa, \chi) = \langle \kappa, \chi : (2{:}\, sold(\chi, \kappa)), (3{:}\, cat(\chi, mc))\rangle$, then the instantiation of r_2 is:

$$(1{:}\, refit) \leftarrow SUM\{\phi(1, p_1), \phi(1, p_2), \dots, \phi(25, p_1), \phi(25, p_2)\} \geq 25.$$

The only equilibrium S' of M_2' is $S' = (\{refit, deliv\,(p_1, 11)\}\,, S_2', S_3')$.

3.3 Complexity

As in ASP, variables in bridge rules allow for an exponentially more succinct representation of MCSs which is reflected in the complexity of respective reasoning tasks. The further addition of aggregates, however, does not increase (worst-case) complexity.

We assume familiarity with well-known complexity classes, restrict to finite MCSs (knowledge bases and bridge rules are finite), and consider logics that have *exponential-size kernels*, i.e., acceptable belief sets S of kb are uniquely determined by a ground subset $K \subseteq S$ of size exponential in the size of kb (cf. also [3] for poly-size kernels).

The context complexity of a context C over logic L is in $\mathcal{C}$, for a complexity class $\mathcal{C}$, iff given kb, a ground belief b, and a set of ground beliefs K, both, deciding whether K is the kernel of some $S \in ACC(kb)$, and deciding whether $b \in S$, is in $\mathcal{C}$.

Theorem 1. *Given a finite relational MCS $M = (C_1, \dots, C_n)$ (with aggregates), where the context complexity is in Δ_k^P, $k \geq 1$, for every context C_i, deciding whether M has an equilibrium and brave reasoning is in* $\text{NEXP}^{\Sigma_{k-1}^P}$*, while cautious reasoning is in* co-$\text{NEXP}^{\Sigma_{k-1}^P}$.

Intuitively, one can non-deterministically guess the kernels $K = (K_1, \ldots, K_n)$ of an equilibrium $S = (S_1, \ldots, S_n)$ together with applicable bridge rule heads (the guess may be exponential in the size of the input) and then verify in time polynomial in the size of the guess, whether S is an equilibrium, resp. whether $b \in S_i$, using the oracle.

Note that the above assumptions apply to various standard KR formalisms which allow for a succinct representation of relational knowledge (e.g., Datalog, Description Logics, etc.) and completeness is obtained in many cases for particular context logics.

4 Grounded Equlibria

In this section we introduce alternative semantics for MCSs with aggregates inspired by grounded equilibria as introduced in [3]. The motivation for considering grounded equilibria is rooted in the observation that equilibria allow for self justification of beliefs via bridge rules, which is also reflected in the observation that in general equilibria are not subset minimal (component-wise).

Grounded equilibria are defined in terms of a reduct for MCSs comprised of so-called reducible contexts. The notions of a *monotonic logic*, *reducible logic*, *reducible context*, *reducible MCS*, and *definite MCS* carry over from standard MCSs [3] to relational MCSs (with aggregates) straightforwardly.

The set of grounded equilibria, denoted by $GE(M)$, of a definite MCS M is the set of component-wise subset minimal equilibria of $grd(M)$. Note that $GE(M)$ is a singleton, i.e., the grounded equilibrium of a definite MCS M is unique, if all aggregate atoms in bridge rules of $grd(M)$ are montone. The latter is the case for a ground aggregate atom a iff $S \models a$ implies $S' \models a$, for all belief states S and S' such that $S_i \subseteq S'_i$ for $1 \leq i \leq n$.

Since equilibria for MCSs do not resort to foundedness or minimality criteria, their extension to MCSs with aggregates is non-ambiguous: it basically hinges on a 'classical' interpretation of ground aggregate atoms wrt. a belief state, independent of a previous 'pre-interpretation'. Due to the potential nonmonotonicity of aggregate atoms however, the situation is different for grounded semantics. This is analogous to the situation in ASP: there is consensus on how to define the (classical) models of ground rules with aggregates, but no mutual consent and different proposals for defining their answer sets.

Three well-known approaches, namely FLP semantics [7], SPT-PDB semantics [14], and Ferraris semantics [8], are briefly restated below for the ASP setting introduced in Section 2. They are either defined, or can equivalently be characterized, in terms of a reduct, which inspires the definition of a corresponding grounded semantics for MCSs with aggregates. For each definition, we provide a correspondence result witnessing correctness: the semantics is preserved when instead of a program P, a single context MCS M_P is considered, where rules of P are self referential bridge rules of M_P.

More formally, we associate a program P with the MCS $M_P = (C_P)$, where L_P is such that $KB_P = BS_P$ given by the power set of the Herbrand Base of P and $ACC_{C_P}(kb) = \{kb\}$, $kb_P = \emptyset$, br_P consists of a bridge rule for each $r \in P$ which is obtained by replacing standard atoms a by bridge atoms $(C_P\colon a)$, and $D_P = (D_{P,P})$ with $D_{P,P}$ denoting the Herbrand Universe of P. Observe that L_P is monotonic.

We illustrate differences of the semantics on the following example.

Example 7 (Lee and Meng [12]). Consider the program P (left) and its associated MCS $M_P = (C)$ with $C = (L_P, \emptyset, br_P, D_P)$ and br_P (right):

$$\begin{array}{rlrl} p(2) \leftarrow & \mathit{not}\, \mathit{SUM}\{X : p(X)\} < 2. & (C{:}p(2)) \leftarrow & \mathit{not}\, \mathit{SUM}\{X : (C{:}p(X))\} < 2. \\ p(-1) \leftarrow & \mathit{SUM}\{X : p(X)\} \geq 0. & (C{:}p(-1)) \leftarrow & \mathit{SUM}\{X : (C{:}p(X))\} \geq 0. \\ p(1) \leftarrow & p(-1). & (C{:}p(1)) \leftarrow & (C{:}p(-1)). \end{array}$$

4.1 FLP Semantics

The FLP-reduct P^I of a ground program P wrt. an interpretation I is the set of rules $r \in P$ such that their body is satisfied wrt. I. An interpretation I is an answer set of P iff it is a $\subseteq$-minimal model of P^I. As usual, semantics for a non-ground program is given by the answer sets of its grounding (also for the subsequent semantics).

Given a ground reducible MCS $M = (C_1, \ldots, C_n)$ with aggregates, and a belief state S, we subsequently introduce reducts $br^{\chi(S)}$ for sets of bridge rules br. The corresponding χ-reduct of M wrt. S is defined as $M^{\chi(S)} = (C_1^{\chi(S)}, \ldots, C_n^{\chi(S)})$, where $C_i^{\chi(S)} = (L_i, red_{L_i}(kb_i, S_i), br_i^{\chi(S)}, D_i)$, for $1 \leq i \leq n$. The *FLP-reduct* of a set of ground bridge rules br wrt. S is the set $br^{FLP(S)} = \{r \in br \mid S \models body(r)\}$.

Definition 5 (FLP equilibrium). *Let M be a reducible MCS with aggregates and S a belief state. S is an FLP equilibrium of M iff $S \in GE(grd(M)^{FLP(S)})$.*

For P in Example 7 the only answer set according to FLP semantics is $S = \{p(-1), p(1)\}$. Correspondingly, (S) is the only FLP equilibrium of M_P.

4.2 SPT-PDB Semantics

Following [14], semantics is defined using the well-known Gelfond-Lifschitz reduct [10] wrt. an interpretation I, denoted $P^{GL(I)}$ here, and applying a monotonic immediate-consequence operator based on conditional satisfaction. Conditional satisfaction of a ground atom a wrt. a pair of interpretations (I, J), denoted by $(I, J) \models a$, is as follows: if a is a standard atom, then $(I, J) \models a$ iff $a \in I$; if a is an aggregate atom then $(I, J) \models a$ iff $I' \models a$, for all interpretations I' such that $I \subseteq I' \subseteq J$. This is extended to conjunctions as usual. The operator τ_P is defined for positive ground programs P, i.e., consisting of rules $a \leftarrow b_1, \ldots, b_k$, and a fixed interpretation J by $\tau_{P,J}(I) = \{a \mid r \in P \wedge (I, J) \models (b_1 \wedge \ldots \wedge b_k)\}$. An interpretation I is an answer set of P under SPT-PDB semantics iff I is the least fixed-point of $\tau_{P^{GL(I)},I}$.

Given a ground reducible MCS $M = (C_1, \ldots, C_n)$ with aggregates, and a belief state S, the *GL-reduct* of a set of ground bridge rules br wrt. S is the set $br^{GL(S)} = \{hd(r) \leftarrow pos(r) \mid r \in br \wedge S \not\models a \text{ for all } a \in neg(r)\}$. Conditional satisfaction carries over to pairs of belief states under component-wise subset inclusion in the obvious way. For a set of ground definite bridge rules br and a pair of belief states (S, T), let $chd(br, S, T) = \{hd(r) \mid r \in br \wedge (S, T) \models body(r)\}$. The operator $\tau_{M,S}$ is defined for a ground definite MCS M and belief state S by $\tau_{M,S}(T) = T'$, where $T' = (T'_1, \ldots, T'_n)$ such that $T'_i = ACC_i\{kb_i \cup chd(br_i, T, S)\}$, for $1 \leq i \leq n$.

Definition 6 (SPT-PDB equilibrium). *Let M be a reducible MCS with aggregates and S a belief state. S is an SPT-PDB equilibrium of M iff S is the least fixed-point of $\tau_{M^{GL(S)},S}$.*

Example 8. Program P has no answer sets under SPT-PDB semantics and M_P has no SPT-PDB equilibrium. Consider, e.g., the FLP equilibrium $(S) = (\{p(-1), p(1)\})$. The reduct $grd(M_P)^{GL(S)}$ is given by the ground instances of bridge rules $r_1 = (C_P{:}\, p(-1)) \leftarrow SUM\{X : (C_P{:}\, p(X))\} \geq 0$ and $r_2 = (C_P{:}\, p(1)) \leftarrow (C_P{:}\, p(-1))$. For $\tau_{grd(M_P)^{GL(S)},S}(\emptyset)$, the ground instance of r_1 is not applicable, because conditional satisfaction does not hold. Also the body of r_2 is not satisfied. Therefore the least-fixed point of $\tau_{grd(M_P)^{GL(S)},S}$ is $\emptyset$, hence S is not an equilibrium.

4.3 Ferraris Semantics

Ferraris semantics [8] has originally been defined for propositional theories under answer set semantics. For our setting, it is characterized by a reduct, where not only rules are reduced, but also the conjunctions $Conj$ of ground sets. Given an interpretation I and a ground aggregate atom $a = f(S) \prec k$ the Ferraris reduct of a wrt. I, denoted $a^{Fer(I)}$ is $f(S') \prec k$, where S' is obtained from S dropping negative standard literals $not\, b$ from $Conj$ if $I \models b$, for every $\langle \bar{t}{:}\, Conj \rangle$ in S. For a positive standard literal a, $a^{Fer(I)}$ is a; for a ground rule $r = a \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_m$, $r^{Fer(I)} = a \leftarrow b_1^{Fer(I)}, \ldots, b_k^{Fer(I)}$. The Ferraris reduct $P^{Fer(I)}$ of a ground program P wrt. an interpretation I is the set of rules $r^{Fer(I)}$ such that $r \in P$ and the body of r is satisfied wrt. I. I is an answer set of P iff it is a $\subseteq$-minimal model of $P^{Fer(I)}$.

Given a ground reducible MCS $M = (C_1, \ldots, C_n)$ with aggregates, and a belief state S, the *Ferraris reduct* of a set of ground bridge rules br wrt. S is the set $br^{Fer(S)} = \{r^{Fer(S)} \mid r \in br \wedge S \models body(r)\}$, where $r^{Fer(S)}$ is the obvious extension of the Ferraris reduct to bridge rules.

Definition 7 (Ferraris equilibrium). *Let M be a reducible MCS with aggregates and S a belief state. Then, S is a Ferraris equilibrium of M iff $S \in GE(grd(M)^{Fer(S)})$.*

Example 9. P has two answer sets under Ferraris semantics: $S_1 = \{p(-1), p(1)\}$ and $S_2 = \{p(-1), p(1), p(2)\}$, intuitively because they yield different reducts. Again both, (S_1) and (S_2), are Ferraris equilibria of M_P.

Proposition 1. *Let P be a program and M_P be its associated MCS, then S is an answer set of P iff (S) is an equilibrium of M_P holds for FLP, SPT-PDB, and Ferraris semantics.*

5 Implementation and Initial Experiments

In this section we present the DMCSAgg system which computes (partial) equilibria of an MCS with aggregates in a distributed way, and briefly discuss initial experiments.

Algorithm 1. $\mathsf{AggEval}(\mathcal{T}, \mathcal{I}_k)$ at $C_k{=}(L_k, kb_k, br_k, D_k)$

```
    Input: $\mathcal{T}$: set of accumulated partial belief states, $\mathcal{I}_k$: set of unresolved neighbours
    Data: $v(c,k)$: relevant interface according to query plan wrt. predecessor $c$
    Output: set of accumulated partial belief states
    if $\mathcal{I}_k \neq \emptyset$ then
(a)     foreach $r \in br_k$ do
        $br_k := br_k \setminus \{r\} \cup \{rewrite(r, Aux)\}$          // rewrite $br_k$
        $\mathcal{T} := \mathsf{guess}(v(c,k) \cup Aux) \bowtie \mathcal{T}$
(b)     foreach $T \in \mathcal{T}$ do $\mathcal{S} := \mathcal{S} \cup \mathsf{lsolve}(T)$      // get local beliefs w.r.t. $T$
(c)     $\mathcal{S} := \{S' \in \mathcal{S} \mid p_{agg_i(r)} \in S' \text{ iff } S' \models agg_i(r)\}$      // check compliance
    else
(b)     foreach $T \in \mathcal{T}$ do $\mathcal{S} := \mathcal{S} \cup \mathsf{lsolve}(T)$      // get local beliefs w.r.t. $T$
    return $\mathcal{S}$
```

Distributed Evaluation Algorithm. DMCSAgg extends of the DMCSOpt algorithm for standard MCSs. We focus on the necessary modifications and describe the underlying ideas informally; for more formal details we refer to [2].

DMCSAgg operates on *partially ground*, relational MCSs with aggregates, i.e., each set of bridge rules is partially ground (cf. Section 3.2). The basic idea of the overall algorithm is that starting from a particular context, a given query plan is traversed until a leaf context is reached. A leaf context computes its local belief sets and communicates back a partial belief state consisting of the projection to a relevant portion of the alphabet, the relevant interface (obtained from corresponding labels of the query plan). When all neighbours of a context have communicated back, then the context can build its own local belief states based on the partial belief states of its neighbours. The partial belief states obtained at the starting context are returned as a result.

Since bridge rules are partially ground, query plan generation works as for DMCSOpt. A first modification concerns subroutine $\mathsf{lsolve}(S)$, responsible for computing local belief states at a context C_k given a partial belief state S. It is modified to first evaluate every aggregate atom in br_k wrt. S. Then, intuitively, each aggregate atom is replaced in br_k according to its evaluation, and the subsequent local belief state computation proceeds as before on the modified (now ground) set of bridge rules.

In our running example from the logistics domain, when leaves C_2 and C_3 have returned S'_2 and S'_3 (cf. Example 6), then the partially ground bridge rule r_2 of C_1 is replaced by the ground bridge rule $(1\colon refit) \leftarrow \top$, since the aggregate evaluates to true. Finally, the expected equilibrium $(\{refit, deliv\,(p_1, 11)\}\,, S'_2, S'_3)$ is returned to the user. We leave a more detailed illustration of the algorithm, also on other application scenarios (in particular cyclic ones, cf. below), for an extended version of the paper.

The second modification of the DMCSOpt algorithm concerns guessing in case of cycles and is described in Algorithm 1 above: If a cycle is detected at context C_k ($\mathcal{I}_k \neq \emptyset$), then (a) it has to be broken by guessing on the relevant interface ($c(v,k)$). To keep guesses small in the presence of aggregates, a local rewriting is employed, such that just the valuation of an aggregate atom is guessed (rather than guessing on the grounding of all atoms in its symbolic set). To this aim, every rule $r \in br_k$ is rewritten, replacing each aggregate $agg_i(r)$ with a new 0-ary predicate $p_{agg_i(r)}$. Then we guess on $c(v,k) \cup Aux$, where *Aux* denotes the set of newly introduced atoms. Next (c), acceptable belief sets

Table 1. DMCSAgg evaluation time in seconds

Top. /(m,r,a)	d=30	d=100	Top. /(m,r,a)	d=15	d=30	Top. /(m,r,a)	d=15	d=30
$D/(16,10,1)$	8.32	51.62	$D/(16,10,2)$	414.62	–	$D/(4,10,3)$	–	–
$Z/(16,10,1)$	19.45	18.19	$Z/(4,10,2)$	486.94	–	$Z/(16,10,3)$	–	–
$B/(16,10,1)$	6.17	10.49	$B/(16,10,2)$	404.26	–	$B/(4,8,3)$	411.00	–

of C_k are computed given the beliefs of its neighbours. If a guess was made on some $p_{agg_i(r)}$, then a test (d) checks every resulting belief state $\mathcal{S}$ for compliance with the guess, i.e., whether $p_{agg_i(r)} \in \mathcal{S}$ iff $\mathcal{S} \models agg_i(r)$, before the computed belief states are returned.

C_k.DMCSAgg(k) denotes a call to DMCSAgg, and correctness and completeness hold:

Theorem 2. *Let $M = (C_1, \ldots, C_n)$ be a (partially) ground, relational MCS with aggregates, C_k a context of M, Π_k a suitable query plan, and let V be a set of ground beliefs of neighbours of C_k. Then, (i) for each $S' \in C_k$.DMCSAgg(k) exists a partial equilibrium S of M wrt. C_k such that $S' = S|_V$, and (ii) for each partial equilibrium S of M wrt. C_k exists an $S' \in C_k$.DMCSAgg(k) such that $S' = S|_V$.*

Initial Experiments. A prototype implementation of DMCSAgg[2] has been developed, which computes partial equilibria of MCSs with ASP as context logics. Like DMCSOpt it applies a loop formula transformation to ASP context programs resulting in SAT instances which are solved using clasp (2010-09-24) [9]. For context grounding *gringo* (2010-04-10) is used, so contexts can be given in the Lparse[15] language; a simple front-end (partially) grounds bridge rules.

Originally, DMCSOpt calls the SAT solver just once per context, guessing for all atoms in bridge rules. In the presence of variables, this causes memory overload due to large guesses even for small domain sizes. We thus applied a naive pushing strategy, i.e., we call the SAT solver once for every partial belief state of the neighbours and push the partial belief state into the SAT formula.

For preliminary experimentation, we adapted a randomized generator that has been used to evaluate DMCSOpt on specific MCS topologies: diamonds, binary trees, zig zag, and ring (where only the last is cyclic, cf. [2] for details). We fixed the following parameters: 10 contexts, with 10 predicates, and at most 4 exported predicates per context; and varied domain sizes d (15, 20, 30, and 100), bridge rules r (8 or 10), arity of relational bridge rule elements a $(1 - 3)$, and the number of local models m (4 or 16). Table 5 lists evaluation time (seconds) for computing the partial belief state of a "root" context on an Intel Core $i5$ 1.73GHz, 6GB RAM with a timeout of 600.

For acyclic topologies and unary predicates, the system scales to larger alphabets than the propositional DMCSOpt system, which can be explained by naive pushing and suggests further optimizations in this direction. However when increasing predicate arities, and for cyclic topologies, limits are reached quickly (for the ring, domain sizes

[2] The system is available in the DMCS repository at
`www.kr.tuwien.ac.at/research/systems/dmcs`

only up to 10 could be handled, which is in line with results on comparable DMCSOpt instances however). Although this is for complexity reasons in general, a naive generation of loop formulas and grounding of bridge rules currently impedes to cope, e.g., with state of the art ASP solvers. Corresponding improvements and specifically more sophisticated cycle breaking techniques are vital (choosing a context with smallest import domain; reducing the relevant interface, i.e. guess size, by taking topology into account).

6 Conclusion

We enhanced the modeling power of MCSs by open bridge rules with variables and aggregates, lifting the framework to relational MCSs with aggregates. In addition to defining syntax and semantics in terms of equilibria and different grounded equilibria, for an implementation we extended the DMCSOpt algorithm to handle the relational setting and aggregates. Initial experiments with the system demonstrate reasonable scaling compared to DMCSOpt but also clearly indicate needs for further optimization.

Besides implementation improvements, topics for future research include the application in real world settings. We envisage its employment in a system for personalized semantic routing in an urban environment. A further interesting application domain are applications where social choices, i.e., group decisions such as voting, play a role. In this respect, our work is remotely related to Social ASP [5], considered as a particular MCS with ASP logics. As for theory, we consider the development of methods for open reasoning that avoid (complete) grounding an interesting and important long-term goal.

References

1. Antoniou, G., Bikakis, A., Papatheodorou, C.: Reasoning with imperfect context and preference information in multi-context systems. In: Catania, B., Ivanović, M., Thalheim, B. (eds.) ADBIS 2010. LNCS, vol. 6295, pp. 1–12. Springer, Heidelberg (2010)
2. Bairakdar, S.E.D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Decomposition of distributed nonmonotonic multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 24–37. Springer, Heidelberg (2010)
3. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: AAAI, pp. 385–390. AAAI Press, Menlo Park (2007)
4. Brewka, G., Roelofsen, F., Serafini, L.: Contextual default reasoning. In: IJCAI, pp. 268–273 (2007)
5. Buccafurri, F., Caminiti, G.: Logic programming with social features. TPLP 8(5-6), 643–690 (2008)
6. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed nonmonotonic multi-context systems. In: KR, pp. 60–70. AAAI Press, Menlo Park (2010)
7. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
8. Ferraris, P.: Answer sets for propositional theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 119–131. Springer, Heidelberg (2005)

9. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI, pp. 386–392 (2007)
10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080 (1988)
11. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics or: How we can do without modal logics. Artif. Intell. 65(1), 29–70 (1994)
12. Lee, J., Meng, Y.: On reductive semantics of aggregates in answer set programming. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 182–195. Springer, Heidelberg (2009)
13. Shen, Y.D., You, J.H., Yuan, L.Y.: Characterizations of stable model semantics for logic programs with arbitrary constraint atoms. TPLP 9(4), 529–564 (2009)
14. Son, T.C., Pontelli, E., Tu, P.H.: Answer sets for logic programs with arbitrary abstract constraint atoms. J. Artif. Intell. Res. (JAIR) 29, 353–389 (2007)
15. Syrjänen, T.: Lparse 1.0 user's manual (2002)

Stepping through an Answer-Set Program*

Johannes Oetsch, Jörg Pührer, and Hans Tompits

Technische Universität Wien, Institut für Informationssysteme 184/3,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. We introduce a framework for interactive stepping through an answer-set program as a means for debugging. In procedural languages, stepping is a widespread and effective debugging strategy. The idea is to gain insight into the behaviour of a program by executing statement by statement, following the program's control flow. Stepping has not been considered for answer-set programs so far, presumably because of their lack of a control flow. The framework we provide allows for stepwise constructing interpretations following the user's intuition on which rule instances to become active. That is, we do not impose any ordering on the rules but give the programmer the freedom to guide the stepping process. Due to simple syntactic restrictions, each step results in a state that guarantees stability of the intermediate interpretation. We present how stepping can be started from breakpoints as in conventional programming and discuss how the approach can be used for debugging using a running example.

Keywords: answer-set programming, program analysis, debugging.

1 Introduction

Answer-set programming (ASP) is a well-established paradigm for declarative problem solving [1], yet it is rarely used by engineers outside academia so far. Arguably, one particular obstacle preventing software engineers from using ASP is the lack of support tools for developing answer-set programs.

In this paper, we introduce a framework that allows for stepping through answer-set programs. Step-by-step execution of a program is folklore in procedural programming languages, where developers can debug and investigate the behaviour of their programs in an incremental way. As answer-set programs have a genuine declarative semantics lacking any control flow, it is not obvious how stepping can be realised. Our approach makes use of a simple computation model that is based on states, which are ground rules that a user considers as active in a program. With each state, we associate the interpretation that is induced by the respective rules. This interpretation is guaranteed to be an answer set of the set of rules considered in the state. During the course of stepping, the interpretations of the subsequent states evolve towards an answer set of the overall program. Our stepping approach is *interactive* and *incremental*, letting the programmer choose which rules are added at each step. In our framework, states may serve

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698 and by the European Commission under project IST-2009-231875 (OntoRule).

as *breakpoints* from which stepping can be started. We discuss how the programmer can generate breakpoints that allow him or her to jump directly to interesting situations. We also show how ground rules that are subsequently considered active can be quickly obtained from the non-ground source code using filtering techniques.

The main area of application of stepping is debugging. A general problem in debugging is to restrict the amount of debugging information that is presented to the user in a sensible way. In the stepping method, this is realised by focussing on one step at a time, which is in contrast to other debugging methods for ASP [2,3,4,5], where the program to be debugged is analysed as a whole. Moreover, due to the interactivity of the approach, the programmer can easily guide the search for bugs following his or her intuitions about which part of the program is likely to be the source of error. Besides debugging, stepping through a program can improve the understanding of the program at hand and can help to improve the understanding of the answer-set semantics for beginners.

The paper is outlined as follows. Section 2 gives the formal background on ASP. The framework for stepping is presented in Section 3. In Section 4, we explain how breakpoints can be generated and how ground rules can be conveniently selected as active ones. Moreover, we describe interesting settings where stepping can be beneficially applied using a running example. After discussing related work in Section 5, we conclude the paper in Section 6.

2 Preliminaries

We deal with *logic programs* which are finite sets of rules of form

$$l_0 \leftarrow l_1, \dots, l_m, \mathrm{not}\; l_{m+1}, \dots, \mathrm{not}\; l_n, \tag{1}$$

where $n \geq m \geq 0$, "not" denotes *default negation*, and all l_i are literals over a function-free first-order language $\mathcal{L}$. A literal is an atom possibly preceded by the *strong negation* symbol $\neg$. For a literal l, we define $\overline{l} = \neg a$ if $l = a$ and $\overline{l} = a$ if $l = \neg a$. In the sequel, we assume that $\mathcal{L}$ will be implicitly given. The literal l_0 may be absent in (1), in which case the rule is a *constraint*. Furthermore , for r of form (1), $\mathrm{B}(r) = \{l_1, \dots, l_m, \mathrm{not}\; l_{m+1}, \dots, \mathrm{not}\; l_n\}$ is the *body* of r, $\mathrm{B}^+(r) = \{l_1, \dots, l_m\}$ is the *positive body* of r, and $\mathrm{B}^-(r) = \{l_{m+1}, \dots, l_n\}$ is the *negative body* of r. The *head* of r is $\mathrm{H}(r) = \{l_0\}$ if l_0 is present and $\mathrm{H}(r) = \emptyset$ otherwise. If $\mathrm{B}(r) = \emptyset$ and $\mathrm{H}(r) \neq \emptyset$, then r is a *fact*. For facts, we usually omit the symbol "$\leftarrow$". Furthermore, we identify sets of literals with sets of facts.

A literal, rule, or program is *ground* if it contains no variables. Let C be a set of constants. A *substitution over* C is a function ϑ assigning each variable in some expression an element of C. We denote by $e\vartheta$ the result of applying ϑ to an expression e. The *grounding* of a program Π, $gr(\Pi)$, is defined as usual.

An *interpretation* I (over some language $\mathcal{L}$) is a finite set of ground literals (over $\mathcal{L}$) such that $\{a, \neg a\} \not\subseteq I$, for any atom a. The satisfaction relation $I \models \alpha$, where α is a ground atom, a literal, a rule, a set of possibly default negated literals, or a program α, is defined in the usual manner. A rule r such that $I \models \mathrm{B}(r)$ is called *active* under I. We denote the set of all active rules of a ground program Π with respect to an interpretation I as $Act^I(\Pi) = \{r \in \Pi \mid I \models \mathrm{B}(r)\}$. Following Faber, Leone, and Pfeifer [6], we define an *answer set* of a program Π as an interpretation I that is a minimal model of $Act^{I}(gr(\Pi))$. For the programs we consider, this definition is equivalent to the traditional one by Gelfond and Lifschitz [7]. The collection of all answer sets of a program Π is denoted by $AS(\Pi)$.

3 Stepping Framework

In this section, we introduce the basic computation model that underlies our stepping approach. We are aiming for a scenario in which the programmer has strong control over the direction of the construction of an answer set. The general idea is to first take a part of a program and an answer set of this part. Then, step by step, rules are added by the user such that, at every step, the literal derived by the new rule is added to the interpretation which remains to be an answer set of the evolving program part. Hereby, the user only adds rules he or she thinks are active in the final answer set. The interpretation grows monotonically until it is eventually guaranteed to be an answer set of the overall program, otherwise the programmer is informed why and at which step something went wrong. This way, one can in principle without any backtracking direct the computation towards an expected or an unintended actual answer set. In debugging, having the programmer in the role of an oracle is a common scenario [8]. It is reasonable to assume that a programmer has good intuitions on where to guide the search if there is a mismatch between the intended and the actual behaviour of a program.

In our framework, the individual steps of a computation—which we regard as *states* of the program—are represented by a set of ground rules which the user considers as active. While these rules represent the state on the source-code level, close to what the programmer has written, we also want to represent a state on the output level in the form of an interpretation that constitutes a partial result of the program. Therefore, we associate a set of ground rules with the interpretation induced by the rules.

Definition 1. *Let S be a set of ground rules. Then, the* interpretation induced by S *is given by $Int[S] = \bigcup_{r \in S} \mathrm{H}(r)$.*

States have to satisfy two properties, ensuring that the interpretation induced by the state is an answer set of the state and that every rule in the state is active with respect to the interpretation. Intuitively, we want every step in the construction of an answer set to result in a stable condition, where we only have rules that are relevant to this condition. The metaphor for that is building up a house of cards, where every card—being the counterpart of a rule—supports the integrity of the evolving house—corresponding to the interpretation—and stability of the house must be ensured after each building activity.

Definition 2. *A set S of ground rules is* self-supporting *if $Int[S] \models \mathrm{B}(r)$, for all $r \in S$, and* stable *if $Int[S] \in AS(S)$. A* state *of a program Π is a set $S \subseteq gr(\Pi)$ of ground rules which is self-supporting and stable.*

Every state can be used as a potential starting point for stepping that allows a programmer to jump directly to an interesting situation, e.g., for debugging purposes.

We next define a successor relation between states and sets of ground rules. The intuition is that a successor of a state S corresponds to a potential state after one step in a computation.

Definition 3. *For a state S of a program Π and a set $S' \subseteq gr(\Pi)$ of ground rules, S' is a* successor *of S in Π, symbolically $S \prec_{\Pi} S'$, if $S' = S \cup \{r\}$, for some rule $r \in gr(\Pi) \setminus S$ with (i) $Int[S] \models \mathrm{B}(r)$, (ii) $\mathrm{H}(r) \neq \emptyset$, and (iii) $\mathrm{H}(r) \cap (\mathrm{B}^{-}(r) \cup \bigcup_{r' \in S} \mathrm{B}^{-}(r') \cup \bigcup_{l \in Int[S]} \bar{l}) = \emptyset$,*

Intuitively, rule r is a rule instance of the program Π that is not yet considered in the current state S but whose preconditions are already satisfied by the state's interpretation, as expressed by Condition (i). Conditions (ii) and (iii) ensure that r is not a constraint and that the literal derived by r is neither inconsistent with $Int[S]$ nor contradicting that all rules in the S' are active. Note that, in general, $Int[S] \subseteq Int[S']$ while $S \subset S'$. The successor relation suffices to "step" from one state to another, i.e., S' is always a state.

Proposition 1. *Let S be a state of a program Π, and $S' \subseteq gr(\Pi)$ a set of ground rules such that $S \prec_{\Pi} S'$. Then, S' is also a state of Π.*

In the following, we define *computations* based on the notion of a state.

Definition 4. *A* computation *for a program Π is a finite sequence $\mathrm{C} = S_0, \ldots, S_n$ of states such that, for all $0 \leq i < n$, $S_i \prec_{\Pi} S_{i+1}$.*

Given a computation $\mathrm{C} = S_0, \ldots, S_n$ for a program Π, in analogy to stepping in procedural programs, we identify the state S_0 at which computation C starts as the *breakpoint* of C. Furthermore, $Int[S_n]$ is called the *result*, $res(\mathrm{C})$, of C.

We next define when a computation has failed, gets stuck, or is complete. Intuitively, failure means that the computation reached a point where no answer set of the program can be reached. A computation is stuck when the last state activated rules deriving literals that are inconsistent with previously chosen active rules. It is considered complete when there are no more unconsidered active rules.

Definition 5. *A computation $\mathrm{C} = S_0, \ldots, S_n$ for Π*

- *has* failed at Step *i if there is no answer set I of Π such that $S_i \subseteq Act^{I}(gr(\Pi))$;*
- *is* stuck *if there is no successor of S_n in Π but there is a rule $r \in gr(\Pi) \setminus S_n$ that is active under $Int[S_n]$;*
- *is* complete *if, for each rule $r \in gr(\Pi)$ that is active under $Int[S_n]$, we have $r \in S_n$.*

The following result guarantees the soundness of our stepping framework.

Theorem 1. *Let Π be a program and $\mathrm{C} = S_0, \ldots, S_n$ a complete computation for Π. Then, $res(\mathrm{C})$ is an answer set of Π.*

The computation model is also complete in the sense that stepping, starting from an arbitrary state S_0 of Π as breakpoint, can reach every answer set $I \supseteq Int[S_0]$ of Π, where $S_0 \subseteq Act^{I}(gr(\Pi))$.

Theorem 2. *Let $I \in AS(\Pi)$ be an answer set of program Π and S_0 a state of Π such that $Int[S_0] \subseteq I$ and $S_0 \subseteq Act^{I}(gr(\Pi))$. Then, there is a complete computation $\mathrm{C} = S_0, \ldots, S_n$ with $S_n = Act^{I}(gr(\Pi))$ and $res(\mathrm{C}) = I$.*

Example 1. Consider the program

$$\begin{aligned}\Pi = \{&obj(c), obj(d), \leftarrow ch(c),\ ch(X) \leftarrow \text{not } \neg ch(X), obj(X),\\ &\neg ch(X) \leftarrow \text{not } ch(X), obj(X)\}.\end{aligned}$$

The answer sets of Π are $I_1 = \{obj(c), obj(d), \neg ch(c), ch(d)\}$ and $I_2 = \{obj(c), obj(d), \neg ch(c), \neg ch(d)\}$. Consider the computation $\mathrm{C} = S_0, S_1, S_2$, where $S_0 = \{obj(c), obj(d)\}$, $S_1 = S_0 \cup \{\neg ch(c) \leftarrow \text{not } ch(c), obj(c)\}$, and $S_2 = S_1 \cup \{ch(d) \leftarrow \text{not } \neg ch(d), obj(d)\}$. Computation C is complete and $res(\mathrm{C}) = I_1$. Consider now the computation $\mathrm{C}' = S'_0, S'_1, S'_2$, where $S'_0 = \{obj(c), obj(d)\}$, $S'_1 = S'_0 \cup \{ch(c) \leftarrow \text{not } \neg ch(c), obj(c)\}$, and $S'_2 = S'_1 \cup \{ch(d) \leftarrow \text{not } \neg ch(d), obj(d)\}$. C' has failed at Step 1 as there is no answer set I of Π such that $Act^I(gr(\Pi))$ contains $ch(c) \leftarrow \text{not } \neg ch(c), obj(c)$. Moreover, C' is stuck as there is no state succeeding S'_2 but $\leftarrow ch(c)$ is active under $Int[S'_2]$.

Observe that once a computation C has failed at some step all computations that contain C as subsequence are guaranteed to get stuck. Hence, when failure is detected at the current step, the user knows that the last active rule chosen is crucial for the targeted interpretation not to be an answer set. Failure of a computation does not mean that it is useless for debugging. In fact, when a program Π does not have any answer set, building up a computation for Π will guide the user to rules responsible for the inconsistency.

The next corollary is a consequence of the fact that the empty set is a state of every program.

Corollary 1. *Let Π be a program and I an answer set of Π. Then, there is a complete computation* $\mathrm{C} = S_0, \ldots, S_n$ *such that* $S_0 = \emptyset$, $S_n = Act^I(gr(\Pi))$, *and* $res(\mathrm{C}) = I$.

The programmer will typically want to start with another breakpoint than the empty set. As we argue in Section 4, obtaining a breakpoint that is "near" to an interesting situation is desirable and, using the programmer's intuition, in most cases not difficult to achieve.

4 Interactive Stepping

In this section, we outline how the framework introduced in the previous section can be used in practice. We will use the *maze-generation problem*, a benchmark problem from the second ASP competition [9], as a running example. The task is to generate a two-dimensional grid where each cell is either a wall or empty. There are two dedicated empty cells located at the border, being the maze's entrance and its exit, respectively. The maze grid has to satisfy the following conditions: (i) except for the entrance and the exit, border cells are walls; (ii) there must be a path from the entrance to every empty cell (including the exit); (iii) if two walls are diagonally adjacent, one of their common neighbours is a wall; (iv) there must not be any 2×2 block of empty cells or walls; and (v) no wall can be completely surrounded by empty cells. The input of this problem are facts that specify the size and the positions of the entrance and the exit of the maze as well as facts that specify for an arbitrary subset of the cells whether they are walls or empty. The output corresponds to completions of the input that represent valid maze structures. As example input, we consider the following set of facts

$$F = \{row(1), row(2), row(3), row(4), row(5), col(1), col(2), col(3), col(4), col(5), \\ entrance(1,2), exit(5,4), wall(3,3), empty(3,4)\}$$

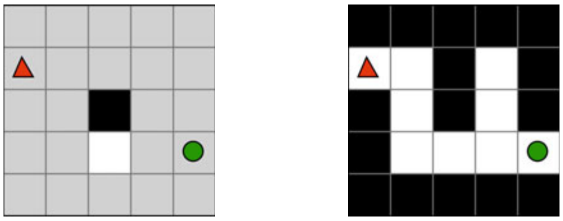


Fig. 1. A maze-generation input with a corresponding solution

that is visualised in Fig. 1 together with a completion to a legal maze. White cells correspond to empty cells, black cells to walls, and grey areas are yet unassigned cells. The entrance is marked by a triangle and the exit by a circle. The program developed in the sequel follows the encoding submitted by the Potassco team[1]. Note that the language used in the example is slightly richer than defined in Section 2. We allow for integer arithmetics and assume a sufficient integer range to be available as constants.

In our approach, we always have two options how to proceed: (i) (re-)initialise stepping and start a computation with a new state as breakpoint, or (ii) extend the current computation by adding a further active rule.

In the remainder of the section, we first describe the technical aspects of how to obtain a breakpoint and how ground rule instances can be chosen. Then, we discuss how stepping can be applied in several situations, including typical debugging scenarios.

4.1 Obtaining a Breakpoint

Having a suitable breakpoint at hand will often allow for finding a bug in just a few steps. As mentioned above, the empty set is a trivial state for every program. Besides that, the set of all facts in a program is also ensured to be a state, except for the practically irrelevant case that a literal and its strong negation are asserted.

Example 2. As a first step for developing the maze-generation encoding, we want to identify border cells. Our initial program is $\Pi_0 = F \cup \Pi_{\mathrm{Bdr}}$, where Π_{Bdr} is given by

$$\begin{aligned}
\mathit{maxcol}(X) &\leftarrow \mathit{col}(X), \text{not } \mathit{col}(X+1),\\
\mathit{maxrow}(Y) &\leftarrow \mathit{row}(Y), \text{not } \mathit{row}(Y+1),\\
\mathit{border}(1,Y) &\leftarrow \mathit{col}(1), \mathit{row}(Y),\\
\mathit{border}(X,Y) &\leftarrow \mathit{row}(Y), \mathit{maxcol}(X),\\
\mathit{border}(X,1) &\leftarrow \mathit{col}(X), \mathit{row}(1),\\
\mathit{border}(X,Y) &\leftarrow \mathit{col}(X), \mathit{maxrow}(Y).
\end{aligned}$$

The first two rules extract the numbers of columns and rows of the maze from the input facts of predicates $\mathit{col}/1$ and $\mathit{row}/1$. The next four rules derive $\mathit{border}/2$ atoms for the grid.

[1] See also `http://dtai.cs.kuleuven.be/events/ASP-competition/Teams/Potassco.shtml`

Now, taking the set F of facts as a breakpoint of a computation for Π_0, we can start stepping by choosing, e.g., the ground rule

$$r = \mathit{maxcol}(5) \leftarrow \mathit{col}(5), \mathrm{not}\ \mathit{col}(6),$$

that is active under F, as next rule to add. We obtain the computation $\mathrm{C} = F, F \cup \{r\}$.

In many cases, it will be useful to have states other than the empty set or the facts as starting points, as starting stepping from them can be time consuming. For illustration, to reach an answer set I of a program, the minimum length of a computation starting from the empty set is $|I|$. We next show how states that may serve as breakpoints can be generated. A state can be obtained by computing an answer set X of a trusted part of a program (or its grounding) and then selecting rule instances that are active under X.

Proposition 2. *Let Π be a program and $\Pi' \subseteq \Pi \cup gr(\Pi)$ such that $I \in AS(\Pi')$. Then, $Act^I(gr(\Pi'))$ is a state of Π.*

Hence, it suffices to find an appropriate Π' in order to get breakpoints. One option for doing so is to let the user manually specify Π' as a subset of Π (including facts).

Example 3. We want to step through the rules that derive the $\mathit{border}/2$ atoms. As we pointed out above, the respective definitions rely on information about the size of the maze. Hence, we will use a breakpoint where the rules deriving $\mathit{maxcol}/1$ and $\mathit{maxrow}/1$ were already applied. Following Proposition 2, we calculate an answer set of program $\Pi_0' \subseteq \Pi_0$ that is given by $\Pi_0' = F \cup \{\mathit{maxcol}(X) \leftarrow \mathit{col}(X), \mathrm{not}\ \mathit{col}(X+1), \mathit{maxrow}(Y) \leftarrow \mathit{row}(Y), \mathrm{not}\ \mathit{row}(Y+1)\}$. The unique answer set of Π_0' is $I_0 = F \cup \{\mathit{maxcol}(5), \mathit{maxrow}(5)\}$. The desired breakpoint S_0 is given by $S_0 = Act^{I_0}(gr(\Pi_0'))$, which consists of the facts in F and the rules $\mathit{maxcol}(5) \leftarrow \mathit{col}(5), \mathrm{not}\ \mathit{col}(6)$ and $\mathit{maxrow}(5) \leftarrow \mathit{row}(5), \mathrm{not}\ \mathit{row}(6)$.

Note that if the subprogram Π' for breakpoint generation has more than one answer set, the selection of the set $I \in AS(\Pi')$ is based on the programmer's intuition, similar to selecting the next rule in stepping.

A different application of Proposition 2 is *jumping* from one state to another by considering further non-ground rules. This makes sense, e.g., in a debugging situation where the user initially started with a breakpoint S that is considered as an early state in a computation. After few steps and reaching state S', the user realises that the computation from S to S' seems to be as intended and wants to proceed at a point where more literals have already been derived, i.e., after applying a selection Π'' of non-ground rules from Π on top of the interpretation $\mathit{Int}[S']$ associated with S'. Then, Π' is given by $\Pi' = S' \cup \Pi''$. Note that, for an arbitrary answer set I of $AS(\Pi')$, it is not ensured that there is a computation of Π starting from S' and ending with the fresh state $Act^I(gr(\Pi'))$. The reason is that there might be rules in S' that are not active under I. If the programmer wants to assure that there is a computation of Π starting from S' and ending with $Act^I(gr(\Pi'))$, Π' can be joined with the set $\mathit{Con}_{S'} = \{\leftarrow \mathrm{not}\ l \mid l \in \mathrm{B}^+(r), r \in S'\} \cup \{\leftarrow l \mid l \in \mathrm{B}^-(r), r \in S'\}$ of constraints.

Jumping from one state to another is also needed if the user wants to skip several steps and considers one or more non-ground rules instead.

Example 4. Assume the program Π_0 has been extended to Π_1 by adding the rule

$$r_{\mathrm{bw}} = \mathit{wall}(X, Y) \leftarrow \mathit{border}(X, Y), \mathrm{not}\ \mathit{entrance}(X, Y), \mathrm{not}\ \mathit{exit}(X, Y)$$

that ensures that every border cell is a wall, except for the entrance and the exit. As $\Pi_0 \subseteq \Pi_1$, the state S_0 of Π_0 is also a state of Π_1. Hence, assume we started stepping Π_1 from S_0 and successively added the rules $\mathit{border}(1, 1) \leftarrow \mathit{col}(1), \mathit{row}(1)$ and $\mathit{border}(1, 2) \leftarrow \mathit{col}(1), \mathit{row}(2)$. Let S_1 be the resulting state, i.e., the union of these rules and S_0. The cells $(1, 1)$ and $(1, 2)$ have been identified as border cells. According to the problem specification, cell $(1, 1)$ should be a wall, and $(1, 2)$ should be empty as it contains the entrance. To test whether our current program realises that, we want to apply the non-ground rule r_{bw} on top of S_1. Therefore, we use Proposition 2 by first computing the answer set I_1 of $\Pi'_1 = S_1 \cap \{r_{\mathrm{bw}}\}$ that is given by $I_1 = I_0 \cup \{\mathit{border}(1, 1), \mathit{border}(1, 2), \mathit{wall}(1, 1)\}$ as expected. The new state is $S_2 = \mathit{Act}^{I_1}(\mathit{gr}(\Pi'_1))$.

4.2 Stepping

To obtain a successor of a given state S of program Π, by Definition 3 we need a rule $r \in \mathit{gr}(\Pi) \setminus S$ with (i) $\mathit{Int}[S] \models \mathrm{B}(r)$, (ii) $\mathrm{H}(r) \neq \emptyset$, and (iii) $\mathrm{H}(r) \cap (\mathrm{B}^-(r) \cup \bigcup_{r' \in S} \mathrm{B}^-(r') \cup \bigcup_{l \in \mathit{Int}[S]} \bar{l}) = \emptyset$. One can proceed in the following fashion: First, a non-ground rule $r \in \Pi$ with $\mathrm{H}(r) \neq \emptyset$ is selected for instantiation. Then, the user assigns constants to the variables occurring in r. Both steps can be assisted by filtering techniques. A stepping system can provide the user with information which non-ground rules in Π have instances that are active under $\mathit{Int}[S]$ but not contained in S. This can be done using ASP itself using *meta-programming* and *tagging transformations* [5,2].

Example 5. The next version, Π_2, of the maze-generation encoding is obtained by joining Π_1 with the following set Π_{guess} of rules:

$$\begin{aligned}
\mathit{wall}(X, Y) &\leftarrow \mathrm{not}\ \mathit{empty}(X, Y), \mathit{col}(X), \mathit{row}(Y),\\
\mathit{empty}(X, Y) &\leftarrow \mathrm{not}\ \mathit{wall}(X, Y), \mathit{col}(X), \mathit{row}(Y),\\
&\leftarrow \mathit{entrance}(X, Y), \mathit{wall}(X, Y),\\
&\leftarrow \mathit{exit}(X, Y), \mathit{wall}(X, Y).
\end{aligned}$$

Program Π_{guess} guesses whether a cell is a wall or empty while assuring that no wall is guessed on an entrance or exit cell. We start the stepping session from breakpoint $S_3 = \mathit{Act}^{I_3}(\mathit{gr}(\Pi_1))$, where $\mathit{AS}(\Pi_1) = \{I_3\}$. Note that the answer set I_3 of Π_1, which is also the interpretation induced by S_3, encodes a situation where the cells from the input as well as the border cells have already been assigned to be a wall or empty (cf. Fig. 2). There are only two non-ground rules in Π_2 that have active ground instances under I_3 that are not yet contained in S_3: $\mathit{wall}(X, Y) \leftarrow \mathrm{not}\ \mathit{empty}(X, Y), \mathit{col}(X), \mathit{row}(Y)$, and $\mathit{empty}(X, Y) \leftarrow \mathrm{not}\ \mathit{wall}(X, Y), \mathit{col}(X), \mathit{row}(Y)$. An advanced source editor may directly highlight the two rules.

A user can also get assistance for a variable assignment. By assigning the variables in r one after the other, the domains of the remaining ones can always be accordingly restricted such that there is still a compatible ground instance of r that is active under

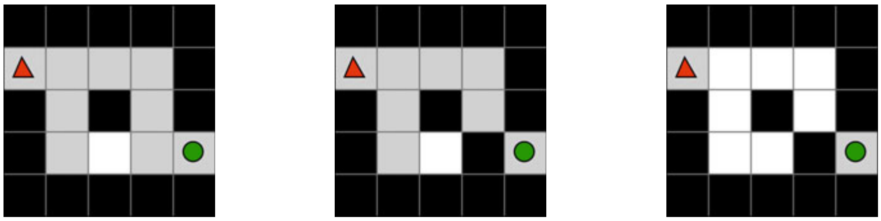


Fig. 2. Visualisation of I_3, I_4, and I_5 from Examples 5, 6, and 9

$Int[S]$. Consider a partial substitution ϑ assigning constants in Π to some variables in r. When fixing the assignment of a further variable X occurring in $\mathrm{B}(r)$, where $\vartheta(X)$ is yet undefined, we may choose only a constant c such that there is a substitution ϑ' with

- $\vartheta'(X') = \vartheta(X')$, where $\vartheta(X')$ is defined,
- $\vartheta'(X) = c$,
- $l\vartheta' \in Int[S]$, for all $l \in \mathrm{B}^+(r)$, and
- $l\vartheta' \notin Int[S]$, for all $l \in \mathrm{B}^-(r)$.

Respective computations can be done by a simple ASP meta-program that guesses ϑ' given r, ϑ, and $Int[S]$, and checks the conditions above plus $r\vartheta' \notin S$.

ASP solvers typically require safety, i.e., all variables occurring in r must also occur in $\mathrm{B}^+(r)$. Thus, the constants to be considered are restricted to those that appear in literals in $Int[S]$ to which literals $\mathrm{B}^+(r)$ can be substituted to. If safety is not given however, all constants in Π have to be considered for the respective substitutions.

Once a substitution ϑ for all variables in r is found, we check whether the newly obtained ground instance $r' = r\vartheta$ satisfies the final condition of Definition 3, i.e., checking whether the head of r' is consistent with all rules in the potential successor state of S being active. If this is not the case, the user's intention that for the considered stepping choices rule r' can be active was wrong.

Example 6. We resume the stepping session of Example 5 at state S_3 and choose rule $r_w = wall(X, Y) \leftarrow \text{not } empty(X, Y), col(X), row(Y)$ for instantiation. There are 24 instances of r_w that are active under I_3. Each such instance corresponds to $r_w\vartheta$, where $\vartheta(X), \vartheta(Y) \in \{1, \ldots, 5\}$ and not both $\vartheta(X) = 3$ and $\vartheta(Y) = 4$. Assume we want to determine the assignment $\vartheta(Y)$ for variable Y first. We choose $\vartheta(Y) = 4$ and determine the value of X next. Filtering now leaves only 1, 2, 4, and 5 as options. We define $\vartheta(X) = 4$ and use the obtained ground instance $r'_w\vartheta = wall(4, 4) \leftarrow \text{not } empty(4, 4), col(4), row(4)$ to step to state $S_4 = S_3 \cup r'_w$. The interpretation $I_4 = Int[S_4]$ is visualised in Fig. 2.

4.3 Application Scenarios

Stepping to an answer set. Stepping until an answer set of a program is reached can be helpful in many situations. Besides the general benefit of getting insights into the interplay of rules of a program, stepping can be used to search for a particular answer set when a program has many of them.

Example 7. Program Π_2 has 128 answer sets and does not yet incorporate all constraints from the problem specification. All answer sets that correspond to valid maze-generation solutions for the given problem instance are among them. Starting stepping from breakpoint S_3, corresponding to the visualisation of I_3 in Fig. 2, we only need nine steps to get to a state S_{sol} where $\mathit{Int}[S_{\text{sol}}]$ encodes the solution depicted in Fig. 1. In fact, we just need to add instances of the rule $\mathit{empty}(X,Y) \leftarrow \text{not } \mathit{wall}(X,Y), \mathit{col}(X), \mathit{row}(Y)$ from Π_{guess} for each unassigned cell (X,Y).

Whenever a state S is reached and $I = \mathit{Int}[S]$ is a desired answer set (projected to interesting literals) of a yet unfinished program, we can make use of the obtained interpretation I for further developing the program. For example, later versions of the program can be tested for being consistent with the intended solution I. If they are not, I can be used as input in a debugging approach, e.g., like the one from earlier work [5] that gives reasons why I is not an answer set of a program.

If the guessing part of a program is extensive, i.e., involving a large number of literals, the guessing rules have to be considered already when obtaining the breakpoint using Proposition 2. If the user has special requirements regarding the guess, for instance that certain cells are not walls, they can be added as constraints when computing the new breakpoint. It is advisable, however, to use small problem instances for testing during program development. Using 5×5 mazes as in our example will be sufficient for realising a reliable encoding and makes stepping easier as computations will be shorter.

Absence of answer sets. A common situation when writing an answer-set program is that the program's current version is unexpectedly incoherent, i.e., it does not yield any answer sets. A usual debugging strategy is to individually remove the constraints of the program to identify which one yields the incoherence. It may be the case that absence of answer sets is not caused by constraints (e.g., contradictory literals, odd loops through negation), but unfortunately, as we will see in the next example, even when the bug is due to constraints, removing constraints is not always sufficient to locate the error.

Example 8. As next features of the maze-generation program, we (incorrectly) implement rules that should express that there has to be a path from the entrance to every empty cell and that 2×2 blocks of empty cells are forbidden. We obtain a new version, Π_3, by joining Π_2 with the rules

$$\begin{aligned}
\mathit{adjacent}(X,Y,X,Y+1) &\leftarrow \mathit{col}(X), \mathit{row}(Y), \mathit{row}(Y+1),\\
\mathit{adjacent}(X,Y,X,Y-1) &\leftarrow \mathit{col}(X), \mathit{row}(Y), \mathit{row}(Y-1),\\
\mathit{adjacent}(X,Y,X+1,Y) &\leftarrow \mathit{col}(X), \mathit{row}(Y), \mathit{col}(X+1),\\
\mathit{adjacent}(X,Y,X-1,Y) &\leftarrow \mathit{col}(X), \mathit{row}(Y), \mathit{col}(X-1),\\
\mathit{reach}(X,Y) &\leftarrow \mathit{entrance}(X,Y), \text{not } \mathit{wall}(X,Y),\\
\mathit{reach}(X_2,Y_2) &\leftarrow \mathit{adjacent}(X_1,Y_1,X_2,Y_2), \mathit{reach}(X_1,Y_1),\\
&\phantom{\leftarrow{}} \text{not } \mathit{wall}(X_2,Y_2),
\end{aligned}$$

formalising when an empty cell is reached from the entrance, and the constraints

$$\begin{aligned}
&c_1 = \leftarrow \mathit{empty}(X,Y), \text{not } \mathit{reach}(X,Y),\\
&c_2 = \leftarrow \mathit{empty}(X,Y), \mathit{empty}(X+1,Y), \mathit{empty}(X,X+1), \mathit{empty}(X+1,Y+1)
\end{aligned}$$

to ensure that every empty cell is reached and that no 2×2 blocks of empty cells exist.

Assume that we did not spot the bug contained in c_2—in the third body literal the term $Y+1$ was mistaken for $X+1$. This could be the result of a typical copy-paste error. It turns out that Π_3 has no answer set. As we already trust the previous version Π_2 and expect the inconsistency to be caused by one of the constraints, the most obvious strategy is to remove one of c_1 or c_2. It turns out that both $\Pi_3 \setminus \{c_1\}$ as well as $\Pi_3 \setminus \{c_2\}$ do have answer sets, 84 answer sets in the former case and ten in the latter case. Inspecting ten answer sets manually is tedious but still manageable. Thus, one could go through them and check whether some of them encode proper maze-generation solutions. After doing so, c_2 would be identified as suspicious. An alternative approach—also feasible if there were more than ten answer sets to expect—is to use the approach of Example 7 and start a computation for Π_3 towards an intended solution, e.g., the one from Fig. 1, at breakpoint S_3. As soon as we reach a state S_{c_2} where the cells $(1,2)$, $(2,1)$ and $(2,3)$ are considered to be empty, there is an instance of constraint c_2 which becomes active with respect to the interpretation induced by S_{c_2}. As noted in Section 4.2, automatic checks after each step could be used to reveal and highlight non-ground rules with active instances. In the case of active constraint instances, it would even be sensible to explicitly warn the programmer. Using the filtering techniques for variable substitutions, the user can be guided to a concrete active instance of c_2, viz. to $c_2' = \leftarrow \mathit{empty}(1,2), \mathit{empty}(2,2), \mathit{empty}(1,2), \mathit{empty}(2,3)$. It is obvious that the cells $(1,2)$,$(2,2)$,$(1,2)$, and $(2,3)$ do not form a valid 2×2 block and hence the wrong term in c_2 can be easily detected. A correct program Π_4 is obtained from Π_3 by changing c_2. As shown in Example 7, depending on the order in which the rules are added, c_2' becomes active in at most nine steps when reaching state S_{sol}.

Understanding someone else's code. Reading and understanding a program written by another developer can be difficult. Stepping through such a program can be quite helpful.

Example 9. Assume we were provided with the code

$$\begin{aligned}
c_3 &= \leftarrow \mathit{wall}(X,Y), \mathit{wall}(X+1,Y), \mathit{wall}(X,Y+1), \mathit{wall}(X+1,Y+1),\\
c_4 &= \leftarrow \mathit{wall}(X,Y), \mathit{wall}(X+1,Y+1), \mathrm{not}\ \mathit{wall}(X+1,Y), \mathrm{not}\ \mathit{wall}(X,Y+1),\\
c_5 &= \leftarrow \mathit{wall}(X+1,Y), \mathit{wall}(X,Y+1), \mathrm{not}\ \mathit{wall}(X,Y), \mathrm{not}\ \mathit{wall}(X+1,Y+1),\\
c_6 &= \leftarrow \mathit{wall}(X,Y), \mathit{empty}(X+1,Y), \mathit{empty}(X-1,Y),\\
&\qquad \mathit{empty}(X,Y+1), \mathit{empty}(X,Y-1),
\end{aligned}$$

implementing the yet uncovered parts of the specification by someone else. Constraint c_3 is similar to the corrected constraint c_2 but forbidding 2×2 blocks of walls instead of empty cells. Constraints c_4 and c_5 ensure that one common neighbour of two diagonally adjacent walls must be a wall. Consequently, the purpose of the remaining constraint c_6 must be to disallow walls to be completely surrounded by empty cells. We are puzzled, however, that the rule already forbids the case that only upper, lower, left, and right neighbours are empty. So, we are wondering how the case that a wall has a single adjacent wall that is the bottom right neighbour is addressed.

To shed light on this issue, we reuse S_4 as a breakpoint for stepping, where $\mathit{Int}[S_4] = I_4$ is illustrated in Fig. 2. We successively add instances of rule $\mathit{empty}(X,Y) \leftarrow \mathrm{not}\ \mathit{wall}(X,Y), \mathit{col}(X), \mathit{row}(Y)$ for fixing the six unassigned cells around the wall at $(3, 3)$ to be empty. Let us assume that S_5 is the state that results ($I_5 = Int[S_5]$ is also depicted in Fig. 2). As the wall in the centre has as its only neighbouring wall the cell $(4, 4)$, we see the requirement that the wall may not be surrounded by empty cells is not violated. Checking for active rules at state S_5 reveals that constraint c_6 has active instances as expected. However, we notice that also constraint c_4 has an active instance under I_5. We now understand why the encoding is correct as the developer of constraints c_3 to c_6 has exploited the interplay of the requirements already that walls without wall neighbours are forbidden and that two diagonally adjacent walls must have one joint neighbour wall. Whenever a wall has only diagonally adjacent walls as neighbours, it is not harmful that constraint c_6 is violated: then, necessarily also the requirement of a common neighbour of the diagonally adjacent walls is violated.

5 Related Work

Previous work on visualising answer-set computations is realised by the noMoRe-system [10]. This system is graph-based and utilises rule dependency graphs (RDGs) which are directed labelled graphs where the nodes are the rules of a given program. Answer sets can be computed by stepwise colouring the nodes of the RDG of a ground program either green or red, reflecting whether a rule is considered active or not. An answer set is formed by the heads of the rules which are coloured green. A handicap for practical stepping is the separation of visualisation from the actual source code due to the graph-based representation and the limitation to ground programs.

Work on debugging in ASP includes *justifications* for ASP [11]. A justification is a labelled directed graph that explains the truth value of a literal with respect to an answer-set in terms of dependency on the truth values of fellow literals. Interesting with respect to our technique is the notion of an *online justification* that explains truth values with respect to partial answer sets emerging during the solving process. As our approach is compatible with the model of computation for online justifications, they can be used in a combined debugging approach. While interactively stepping through a computation allows for following individual intuitions concerning rule applications, justifications or related concepts could keep track of the chosen support for individual literals of interest. A potential shortcoming concerning the intuition of justifications is the absence of program rules, constituting the actual source code artifacts, in the graphs.

Other work on declarative debugging centred on the question why a given interpretation is not an answer set of a program [5]. The answers are given in terms of rule instances that are unsatisfied or loops that are unfounded. As noted in Section 4.2, the meta-programming techniques used in that work allow for identifying active rules in the stepping approach. Also, the interpretation needed as input in that debugging approach could be partially constructed by means of stepping. Note that unfounded loops cannot occur in the stepping process as states are required to be stable.

Another related approach is to trace the concrete execution of a solver. A respective system developed for DLV [12] is intended for debugging the solver itself rather than the answer-set programs. A disadvantage of solver-based tracing for debugging and program analysis is that some solver algorithms do not work on the rule level, are quite involved, and hard to grasp for an ordinary programmer. Interpreted as a strategy for computing answer sets, our stepping model is similar in spirit to a non-deterministic algorithm due to Iwayama and Satoh [13]. Moreover, Gebser et al. [14] introduced an incremental semantics for logic programs based on ι-answer sets. These are relaxations of answer sets that are not necessarily models of the overall program and can be constructed by step-by-step applying active rules similar as in our approach. Their semantics guarantees to reach a ι-answer set, while under standard semantics, computations may fail.

6 Conclusion

We presented a framework for stepping through an answer-set program that is useful for debugging and program analysis. It allows the programmer to follow his or her intuitions regarding which rules to apply next and is based on an intuitive and simple computation model where rules are subsequently added to a state. Every state implicitly defines an interpretation that is stable with respect to that state. We also discussed how to obtain states that may serve as breakpoints from which stepping is started. Keeping a handful of these breakpoints during program development, the programmer can quickly initiate stepping sessions from situations he or she is already familiar with. A prototypical stepping system will be part of SeaLion, an integrated development environment for ASP we are currently developing. In future work, we plan to extend the approach to programs with function symbols, aggregates (possibly in rule heads), and disjunctions.

References

1. Gelfond, M., Leone, N.: Logic programming and knowledge representation - the A-Prolog perspective. Artificial Intelligence 138(1-2), 3–38 (2002)
2. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
3. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proc. NMR 2006, pp. 77–83 (2006)
4. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Proc. AAAI 2008, pp. 448–453. AAAI Press, Menlo Park (2008)
5. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: Towards debugging non-ground answer-set programs. Theory and Practice of Logic Programming 10(4-6), 513–529 (2010)
6. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
7. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3-4), 365–386 (1991)
8. Shapiro, E.Y.: Algorithmic Program Debugging. PhD thesis, Yale University, New Haven, CT, USA (May 1982)
9. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
10. Bösel, A., Linke, T., Schaub, T.: Profiling answer set programming: The visualization component of the noMoRe system. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 702–705. Springer, Heidelberg (2004)

11. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming 9(1), 1–56 (2009)
12. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Proc. SEA 2009 (2009)
13. Iwayama, N., Satoh, K.: Computing abduction by using TMS with top-down expectation. Journal of Logic Programming 44(1-3), 179–206 (2000)
14. Gebser, M., Gharib, M., Mercer, R.E., Schaub, T.: Monotonic answer set programming. Journal of Logic and Computation 19(4), 539–564 (2009)

Dynamic Magic Sets for Programs with Monotone Recursive Aggregates

Mario Alviano, Gianluigi Greco, and Nicola Leone

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{alviano,ggreco,leone}@mat.unical.it

Abstract. Disjunctive Logic Programming (DLP) is an extension of Datalog that allows for disjunction in rule head and nonmonotonic negation in bodies. All of the queries in the second level of the polynomial hierarchy can be expressed in this language. However, DLP does not allow for representing properties which involve sets of data in a natural way. Extending the language by introducing aggregate functions has been proposed in the literature to overcome this lack, then leading to the language DLP$^{\mathcal{A},\neg}$. In particular, DLP$^{\mathcal{A},\neg}$ allows for using recursive aggregates, which naturally arise in many practical application scenarios. An aggregate is recursive if its aggregate set depends on the evaluation of the aggregate itself. The evaluation of programs with aggregates is hard, especially when aggregates are recursive, optimization techniques are highly needed to make these programs usable in real-world applications.

In this paper, we focus on the optimization of queries over programs with recursive aggregates. In particular, we design an extension of the Dynamic Magic Set (DMS) technique to programs with stratified negation and monotone recursive aggregates, and we demonstrate the correctness of the proposed technique. For assessing the effectiveness of the new technique, we consider a standard benchmark for recursive aggregates, referred to as Company Controls, along with a couple of benchmarks involving aggregates over the WordNet database. Experimental results confirm the effectiveness of our technique.

Keywords: Disjunctive Logic Programming, recursive aggregates, Magic Sets.

1 Introduction

Disjunctive Logic Programming (DLP) is a language that has been proposed for modeling incomplete data [1]. Together with a light version of negation, in this paper stratified negation, this language can in fact express any query of the complexity class Σ_2^P [2], under the stable model semantics. For this reason, it is not surprising that DLP has found several practical applications, including team-building [3], semantic-based information extraction [4] and e-tourism [5], also encouraged by the availability of some efficient inference engines, such as DLV [6], GnT [7], Cmodels [8], or ClaspD [9]. As a matter of fact, these systems are continuously enhanced to support novel optimization strategies, enabling them to be effective over increasingly larger application domains.

Despite its expressive power, DLP does not allow for representing in a natural way properties that involve sets of elements, such as properties based on counting the number of elements satisfying user-defined criteria. In fact, extending the language by introducing aggregate functions has been an active area of research in the last few years (see, e.g., [10,11,12,13,14]). In this paper, we consider DLP$^{\mathcal{A}_m,\neg_s}$, which is an extension of DLP allowing the use of stratified negation combined with aggregate functions. These functions must be *monotone*, even though possibly *recursive*, i.e., the sets on which they are applied can even depend on the result of their evaluation[1]. Recursive aggregates naturally arise in many practical application scenarios; for instance, recursive aggregates can be used for consistent query answering over heterogeneous data sources [15].

Designing and implementing inference engines for DLP$^{\mathcal{A}_m,\neg_s}$ is clearly more challenging compared to the case of plain disjunctive logic programs. Indeed, in order to be efficient, inference engines must exploit optimization strategies specifically conceived to deal with aggregate functions. However, this issue has been just marginally faced in earlier literature. In this paper, we fill the gap and propose an optimization method for DLP$^{\mathcal{A}_m,\neg_s}$ which is inspired by deductive database optimization techniques, in particular the Magic Set method [16,17,18].

The Magic Set method is a strategy for simulating the top-down evaluation of a query by modifying the original program by means of additional rules, which narrows the computation to what is relevant for answering the query. Basically, the method propagates the constants in the query to all head rules that unify with the query. Atoms in these rules are taken as subqueries and the procedure is iterated, like SLD–resolution [19]. In particular, if the (sub)query has some arguments *bound* to constant values, this information is "passed" to the atoms in the body. Moreover, bodies are processed in a certain sequence, and processing a body atom may bind some of its arguments for subsequently considered body atoms, thus "generating" and "passing" bindings. The specific propagation strategy adopted to select the order according to which body atoms have to be processed is called *sideways information passing strategy* (SIPS).

The Magic Set method has been originally defined for non-disjunctive programs (see [16]). Subsequently, it has been extended to the DLP language by [20,21], and recently to non-disjunctive programs with unstratified negation [22]. However, up to now, there was no proposal to extend the method to DLP$^{\mathcal{A}_m,\neg_s}$ programs. This is precisely the goal of this paper, the contributions of which are as follows:

- ▷ We show how classical sideways information passing strategies can be modified as to take care of aggregate functions. Based on our novel SIPS, we design an extension of the Magic Set algorithm for DLP$^{\mathcal{A}_m,\neg_s}$ programs.
- ▷ We prove that the evaluation method is sound and complete. Thus, no answer can be missed, while computation is narrowed to the part of the instantiation that is really relevant to answer the query.
- ▷ To assess the efficiency of the proposed technique, we carry out some experiments. The results give a clear evidence of the benefit of the proposed optimization method.

Organization. The remainder of the paper is organized as follows. Section 2 presents an overview of the language DLP$^{\mathcal{A}_m,\neg_s}$. The Magic Set method for this language is then illustrated in Section 3. The correctness of the approach is formally proven in Section 4.

[1] Note that the results in this paper hold also in presence of arbitrary (possibly nonmonotone) non-recursive aggregates, which can be rewritten by using auxiliary rules, monotone aggregates and stratified negation.

Experimental results are discussed in Section 5, and a few final remarks are reported in Section 6. Due to space limitations, details on algorithms and full proofs are reported in Appendix (`http://archives.alviano.com/LPNMR2011/`).

2 The DLP$^{\mathcal{A}_m,\neg_s}$ Language

In this section we present the basis of DLP$^{\mathcal{A},\neg}$— an extension of Disjunctive Logic Programming (DLP) by set–oriented functions, also called aggregate functions. In this section we also introduce the DLP$^{\mathcal{A}_m,\neg_s}$ fragment, which is the language considered in this paper. For further background on DLP, we refer to [23].

Syntax. We assume sets of *variables*, *constants*, *predicates* and *aggregate function symbols* to be given. A *term* is either a variable or a constant. A *standard atom* is an expression $\mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_k)$,[2] where p is a *predicate* of arity $k \geq 0$ and $\mathtt{t}_1,\ldots,\mathtt{t}_k$ are terms. An *aggregate function* is of the form $f(S)$, where f is an aggregate function symbol and S is a set term; a set term is a pair $\{\bar{\mathtt{t}} : \alpha\}$, where $\bar{\mathtt{t}}$ is a list of terms (variables or constants) and α is a standard atom. An *aggregate atom* is a structure of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq\}$ is a comparison operator and T is a term (variable or constant). A *literal* is either (i) a standard atom (a positive literal), (ii) a standard atom preceded by the *negation as failure* symbol `not` (a negative literal) or (iii) an aggregate atom (an aggregate literal).

A *program* is a set of *rules* r of the form $\alpha_1 \;\mathtt{v}\; \cdots \;\mathtt{v}\; \alpha_m \;\text{:-}\; \ell_1,\, \ldots,\, \ell_n$, where $\alpha_1,\, \ldots,\, \alpha_m$ are standard atoms, $\ell_1,\, \ldots,\, \ell_n$ are literals, $m \geq 1$ and $n \geq 0$. The disjunction $\alpha_1 \;\mathtt{v}\; \cdots \;\mathtt{v}\; \alpha_n$ is referred to as the *head* of r and the conjunction $\ell_1,\, \ldots,\, \ell_n$ is the *body* of r. We use $H(r)$ for denoting the set of head atoms, $B^+(r)$ and $B^-(r)$ for the set of atoms appearing in positive and negative body literals, respectively, and $B^{\mathcal{A}}(r)$ for the set of aggregate body literals. The set of standard atoms appearing in a rule r, including standard atoms occurring in aggregate literals, is denoted by $\mathrm{ATOMS}(r)$.A variable appearing solely in sets terms of r is *local*; otherwise, it is *global*. A structure (atom, literal, rule or program) without global variables is *ground*. A ground rule r such that $|H(r)| = 1$ is a *fact*. A predicate p is an *extensional database predicate* (EDB predicate) if all rules r such that p appears in $H(r)$ are facts. A predicate p is an intentional database predicate (IDB predicate) if p is not an EDB predicate.

Semantics. A rule r is *safe* if the following conditions hold: (i) all global variables of r also appear in $B^+(r)$; (ii) each local variable of r appearing in a set term $\{\bar{\mathtt{t}} : \alpha\}$ also appears in α. A program is safe if all of its rules are safe. In this paper, we will consider only safe programs. The *universe* of a DLP$^{\mathcal{A},\neg}$ program $\mathcal{P}$, denoted by $\mathcal{U}_{\mathcal{P}}$, is the set of constants appearing in $\mathcal{P}$.[3] The *base* of $\mathcal{P}$, denoted by $\mathcal{B}_{\mathcal{P}}$, is the set of standard atoms constructible from predicates of $\mathcal{P}$ with constants in $\mathcal{U}_{\mathcal{P}}$. A *substitution* is a mapping from a set of variables to $\mathcal{U}_{\mathcal{P}}$. Given a substitution ϑ and a DLP$^{\mathcal{A},\neg}$ object *obj* (literal, rule, etc.), we denote by *obj* ϑ the object obtained by replacing each variable X in *obj*

[2] We use the notation $\bar{\mathtt{t}}$ for a sequence of terms. Thus, an atom with predicate p is usually referred to as $\mathtt{p}(\bar{\mathtt{t}})$ in this paper.

[3] If $\mathcal{P}$ has no constants, an arbitrary constant is added to $\mathcal{U}_{\mathcal{P}}$.

by $\vartheta(X)$. If $obj\vartheta$ is ground, $obj\vartheta$ is an instance of obj. The set of instances of all rules in a program $\mathcal{P}$ is denoted by $Ground(\mathcal{P})$.

A *partial interpretation* for a DLP$^{\mathcal{A},\neg}$ program $\mathcal{P}$ is a pair $\langle T, U\rangle$ such that $T \subseteq U \subseteq \mathcal{B}_{\mathcal{P}}$. Intuitively, atoms in T are true, atoms in $U \setminus T$ are undefined and atoms in $\mathcal{B}_{\mathcal{P}} \setminus U$ are false. If $T = U$, $\langle T, U\rangle$ is a *total interpretation*, and will be referred to as T. Negative literals are interpreted as follows: (i) $\mathtt{not}\ \alpha$ is true whenever α is false; (ii) $\mathtt{not}\ \alpha$ is undefined whenever α is undefined; (iii) $\mathtt{not}\ \alpha$ is false whenever α is true.

An interpretation also provides a meaning to set terms, aggregate functions and aggregate literals, namely a multiset, a value, and a truth value, respectively. We first consider a total interpretation I. The evaluation $I(S)$ of a set term $S = \langle \bar{\mathtt{t}} : \alpha\rangle$ w.r.t. I is the multiset $I(S)$ defined as follows: Let $S^I = \{\langle \bar{\mathtt{t}}\vartheta\rangle \mid \vartheta$ is a substitution and $\alpha\vartheta \in I\}$, then $I(S)$ is the multiset obtained as the projection of the tuples of S^I on their first constant, that is, $I(S) = [\mathtt{t}_1 \mid \langle \mathtt{t}_1, \ldots, \mathtt{t}_n\rangle \in S^I]$. The evaluation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$ (where $\bot$ is a fixed symbol not occurring in $\mathcal{P}$). A ground aggregate atom $f(S) \prec k$ is true w.r.t. I if both (i) $I(f(S)) \neq \bot$ and (ii) $I(f(S)) \prec k$ hold; otherwise, $f(S) \prec k$ is false. We now consider a *partial* interpretation $\langle T, U\rangle$ and define an *extension* of $\langle T, U\rangle$ as an interpretation $\langle T', U'\rangle$ such that $T \subseteq T'$ and $U' \subseteq U$[4]. If a ground aggregate literal ℓ is true (resp. false) w.r.t. *each total* interpretation I extending $\langle T, U\rangle$, then ℓ is true (resp. false) w.r.t. $\langle T, U\rangle$; otherwise, ℓ is undefined.

Given a total interpretation I, a ground rule r is *satisfied* w.r.t. I if at least one head atom is true w.r.t. I whenever all body literals are true w.r.t. I. A total interpretation M is a *model* of a DLP$^{\mathcal{A},\neg}$ program $\mathcal{P}$ if all the rules in $Ground(\mathcal{P})$ are satisfied w.r.t. M. The semantics of a DLP$^{\mathcal{A},\neg}$ program $\mathcal{P}$ is given by the set of its stable models, denoted by $\mathcal{SM}(\mathcal{P})$. Stable models are defined in terms of a reduct [24]: Given a DLP$^{\mathcal{A},\neg}$ program $\mathcal{P}$ and a total interpretation I, let $Ground(\mathcal{P})^I$ denote the transformed program obtained from $Ground(\mathcal{P})$ by deleting all rules in which a body literal is false w.r.t. I. A total interpretation M is a stable model of $\mathcal{P}$ if M is a subset–minimal model of $Ground(\mathcal{P})^M$.

The DLP$^{\mathcal{A},\neg}$ language also supports queries, which can be associated with brave or cautious reasoning. For a DLP$^{\mathcal{A},\neg}$ program $\mathcal{P}$, a ground atom α is a *brave* (resp. *cautious*) *consequence* of $\mathcal{P}$ if α belongs to some (resp. all) stable model of $\mathcal{P}$. Queries are expressed by atoms, that is, a query $\mathcal{Q}$ is of the form $\mathtt{p}(\bar{\mathtt{t}})$?[5]. The set of substitutions ϑ for the variables of $\mathtt{p}(\bar{\mathtt{t}})$ such that $\mathtt{p}(\bar{\mathtt{t}})\vartheta$ is a brave (resp. cautious) consequence of $\mathcal{P}$ is denoted by $Ans_b(\mathcal{Q}, \mathcal{P})$ (resp. $Ans_c(\mathcal{Q}, \mathcal{P})$)[6]. Two DLP$^{\mathcal{A},\neg}$ programs $\mathcal{P}$ and $\mathcal{P}'$ are *brave* (resp. *cautious*) *equivalent* w.r.t. a query $\mathcal{Q}$, denoted by $\mathcal{P} \equiv^b_{\mathcal{Q}} \mathcal{P}'$ (resp. $\mathcal{P} \equiv^c_{\mathcal{Q}} \mathcal{P}'$), if $Ans_b(\mathcal{Q}, \mathcal{P} \cup \mathcal{F}) = Ans_b(\mathcal{Q}, \mathcal{P}' \cup \mathcal{F})$ (resp. $Ans_c(\mathcal{Q}, \mathcal{P} \cup \mathcal{F}) = Ans_c(\mathcal{Q}, \mathcal{P}' \cup \mathcal{F})$) is guaranteed for each set of facts $\mathcal{F}$ defined over the EDB predicates of $\mathcal{P}$ and $\mathcal{P}'$.

[4] This definition of extension of an interpretation preserves "knowledge monotonicity": All literals which are true (resp. false) w.r.t. $\langle T, U\rangle$ are true (resp. false) w.r.t. $\langle T', U'\rangle$.

[5] For simplicity, question marks of queries will be usually omitted when referring to them in the text. We assume that each constant appearing in $\mathcal{Q}$ also appears in $\mathcal{P}$. Note also that more complex queries are expressible by using additional rules.

[6] For ground queries, these sets are either empty or just contain ϵ (the empty substitution).

Restrictions and Bottom–up Instantiation. The DLP$^{\mathcal{A}_m,\neg_s}$ fragment is defined by means of two restrictions discussed in this section. The first restriction applies to aggregates. Let $\leq$ be a partial order for (partial) interpretations such that $\langle T, U\rangle \leq \langle T', U'\rangle$ if and only if $T \subseteq T'$ and $U \subseteq U'$. A ground aggregate literal ℓ is *monotone* if, for all pairs $\langle T, U\rangle$, $\langle T', U'\rangle$ such that $\langle T, U\rangle \leq \langle T', U'\rangle$, we have that: (i) ℓ true w.r.t. $\langle T, U\rangle$ implies ℓ true w.r.t. $\langle T', U'\rangle$, and (ii) ℓ false w.r.t. $\langle T', U'\rangle$ implies ℓ false w.r.t. $\langle T, U\rangle$. The second restriction applies to negation. A predicate $\mathtt{p}$ appearing in the head of a rule r *depends* on each predicate $\mathtt{q}$ such that an atom $\mathtt{q}(\bar{\mathtt{s}})$ occurs in the body of r; if $\mathtt{q}(\bar{\mathtt{s}})$ belongs to $B^-(r)$, $\mathtt{p}$ depends on $\mathtt{q}$ negatively. A program is *stratified* if recursive dependencies do not involve negative dependencies. Let DLP$^{\mathcal{A}_m,\neg_s}$ denote the set of stratified DLP$^{\mathcal{A},\neg}$ programs in which all aggregates are monotone.

Semantics of DLP$^{\mathcal{A}_m,\neg_s}$ programs and queries can be computed by implementing a two-phase strategy. The first phase, *program instantiation*, associates an input program $\mathcal{P}$ with a ground program which is equivalent to $Ground(\mathcal{P})$, but significantly smaller. Most of the techniques used in this phase stem from bottom–up methods developed for classic and deductive databases; see for example [25] for details. A fact which is used in these techniques is that the truth of an atom $\mathtt{p}(\bar{\mathtt{t}})$ has to be supported by some rule having $\mathtt{p}(\bar{\mathtt{t}})$ in the head and such that all body literals are true. A simple algorithm may start by storing all facts of a DLP$^{\mathcal{A}_m,\neg_s}$ program $\mathcal{P}$ in a set R, also computing the set $\mathcal{H}$ of all atoms occurring in the head of some rule in R. After that, each rule $r \in \mathcal{P}$ may be instantiated w.r.t. all substitutions ϑ such that no literal in $B^+(r)\vartheta \cup B^{\mathcal{A}}(r)\vartheta$ is false w.r.t. $\langle \emptyset, \mathcal{H}\rangle$. All new instances are added to R and the process is repeated until no new rules are produced. The resulting ground program constitutes the input of the second phase, referred to as *stable model search*, which computes stable models (for programs) or substitution answers (for queries); details can be found in [26,27].

3 Magic Sets for DLP$^{\mathcal{A}_m,\neg_s}$ Programs

Dynamic Magic Sets (DMS) are an extension of the original Magic Set technique proposed to optimize disjunctive Datalog programs with stratified negation [28]. The goal of this section is to extend DMS in order to deal with arbitrary DLP$^{\mathcal{A}_m,\neg_s}$ programs.

The DMS algorithm is reported in Fig. 1. DMS starts with a query $\mathcal{Q}$ over a DLP$^{\mathcal{A}_m,\neg_s}$ program $\mathcal{P}$ and outputs a rewritten program DMS$(\mathcal{Q}, \mathcal{P})$. The method uses two sets, S and D, to store adorned predicates to be propagated and already processed, respectively. Magic rules are stored in the set $R^{mgc}_{\mathcal{Q},\mathcal{P}}$, modified rules in $R^{mod}_{\mathcal{Q},\mathcal{P}}$. Initially, all sets S, D, $R^{mgc}_{\mathcal{Q},\mathcal{P}}$ and $R^{mod}_{\mathcal{Q},\mathcal{P}}$ are empty (line 1). The algorithm starts by processing the query (line 2), also putting the adorned version of the query predicate into S. The main loop of the algorithm is then repeated until S is empty (lines 3–10). In particular, an adorned predicate $\mathtt{p}^\alpha$ is moved from S to D (line 4) and each rule r having and atom $\mathtt{p}(\bar{\mathtt{t}})$ in head is considered (lines 5–9). The adorned version r^a of the rule r is computed (line 6), from which magic rules are generated (line 7) and a modified rule r' is obtained (line 8). Finally, the algorithm terminates returning the program obtained by the union of $R^{mgc}_{\mathcal{Q},\mathcal{P}}$, $R^{mod}_{\mathcal{Q},\mathcal{P}}$ and EDB$(\mathcal{P})$ (line 11). A brief description of the four auxiliary functions in Fig. 1 is given below. We will use the following running example.

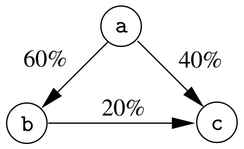
```
Algorithm DMS($\mathcal{Q}$,$\mathcal{P}$)
Input: A query $\mathcal{Q}$ and a DLP$^{\mathcal{A}_m,\neg_s}$ program $\mathcal{P}$
Output: A rewritten program
var
  $S$, $D$ : set of adorned predicates;   $R^{mgc}_{\mathcal{Q},\mathcal{P}}$, $R^{mod}_{\mathcal{Q},\mathcal{P}}$: set of rules;   $r^a$: adorned rule;
begin
   1.  $S := \emptyset$;  $D := \emptyset$;  $R^{mgc}_{\mathcal{Q},\mathcal{P}} := \emptyset$;  $R^{mod}_{\mathcal{Q},\mathcal{P}} := \emptyset$;
   2.  ProcessQuery($\mathcal{Q}$, $S$, $R^{mgc}_{\mathcal{Q},\mathcal{P}}$);
   3.  while $S \neq \emptyset$ do
   4.     take an element $\mathtt{p}^\alpha$ from $S$;   remove $\mathtt{p}^\alpha$ from $S$;   add $\mathtt{p}^\alpha$ to $D$;
   5.     for each rule $r$ in $\mathcal{P}$ and for each atom $\mathtt{p}(\bar{\mathtt{t}})$ in $H(r)$ do
   6.        $r^a$ := Adorn($r$, $\alpha$, $S$, $D$);
   7.        $R^{mgc}_{\mathcal{Q},\mathcal{P}} := R^{mgc}_{\mathcal{Q},\mathcal{P}} \cup$ Generate($r$, $\alpha$, $r^a$);
   8.        $R^{mod}_{\mathcal{Q},\mathcal{P}} := R^{mod}_{\mathcal{Q},\mathcal{P}} \cup \{$Modify($r$, $r^a$)$\}$;
   9.     end for
  10.  end while
  11.  return $R^{mgc}_{\mathcal{Q},\mathcal{P}} \cup R^{mod}_{\mathcal{Q},\mathcal{P}} \cup$ EDB($\mathcal{P}$);
end.
```

Fig. 1. Dynamic Magic Sets algorithm for DLP$^{\mathcal{A}_m,\neg_s}$ programs

Example 1 (Company Controls). Given a set of companies, each of which can own a percentage of shares of the other, a company x exerts control on a company y if x controls, directly or indirectly, more than 50% of shares of y. A simple scenario is depicted in Fig. 2 and represented by the following EDB: $\mathcal{F}_1 = \{\mathtt{comp(a)}, \mathtt{comp(b)}, \mathtt{comp(c)}, \mathtt{owns(a,b,60)}, \mathtt{owns(a,c,40)}, \mathtt{owns(b,c,20)}\}$. In this case, a controls b and, thanks to the 20% of c possessed by b, also c. This problem is known as *Company Controls* and requires to calculate the sum of controlled shares, which in turn depend on the evaluation of this sum. A DLP$^{\mathcal{A}_m,\neg_s}$ program $\mathcal{P}_1$ encoding Company Controls is shown in Fig. 2. Controlled shares are determined by r_1 (directly controlled shares) and r_2 (indirectly controlled shares)[7]. Controls between companies are determined by r_3. Given $\mathcal{P}_1$, a query $\mathcal{Q}_1 = \mathtt{ctrls(a,c)}$ can be used for checking whether company $\mathtt{a}$ exerts control on company $\mathtt{c}$. $\triangleleft$

Function 1: ProcessQuery. For a query $\mathcal{Q} = \mathtt{p}(\bar{\mathtt{t}})$, the function **ProcessQuery** builds an adornment string α for the predicate $\mathtt{p}$. The element in position i of α is b if the i–th argument of $\mathtt{p}(\bar{\mathtt{t}})$ is a constant, otherwise the element in position i of α is f. After that, the adorned predicate $\mathtt{p}^\alpha$ is added to S in order to be subsequently processed for binding propagation. Moreover, the function builds and add to $R^{mgc}_{\mathcal{Q},\mathcal{P}}$ a query seed $\mathtt{mgc}(\mathtt{p}^\alpha(\bar{\mathtt{t}}))$. The function $\mathtt{mgc}(\cdot)$ associates an adorned atom $\mathtt{p}^\alpha(\bar{\mathtt{t}})$ with its magic version $\mathtt{mgc\_p}(\bar{\mathtt{t}}')$, where $\mathtt{mgc\_p}$ is a predicate symbol not occurring in $\mathcal{P}$ and $\bar{\mathtt{t}}'$ is the list of constants corresponding to bound arguments in $\mathtt{p}^\alpha(\bar{\mathtt{t}})$.

[7] Note that r_1 contains a built–in atom $\mathtt{Z} = \mathtt{X}$, that is, we assume that facts of the form $\xi = \xi$ (where ξ is a constant) are contained in all EDB.



$$
\begin{aligned}
r_1 &: \ \mathtt{cs(X,Z,Y,S)} \ \text{:--} \ \mathtt{owns(X,Y,S)},\ \mathtt{Z} = \mathtt{X}.\\
r_2 &: \ \mathtt{cs(X,Z,Y,S)} \ \text{:--} \ \mathtt{owns(Z,Y,S)},\ \mathtt{ctrls(X,Z)}.\\
r_3 &: \ \mathtt{ctrls(X,Y)} \ \text{:--} \ \mathtt{comp(X)},\ \mathtt{comp(Y)},\\
&\qquad\qquad \#\mathtt{sum}\{\mathtt{S,Z} : \ \mathtt{cs(X,Z,Y,S)}\} > 50.
\end{aligned}
$$

Fig. 2. Company Controls: an instance (left) and a DLP$^{\mathcal{A}_m,\neg_s}$ encoding (right)

Example 2. For the query $\mathcal{Q}_1$ from Example 1, **ProcessQuery** builds the adorned predicate $\mathtt{ctrls}^{bb}$ and the query seed $r_{\mathcal{Q}_1}: \ \mathtt{mgc\_ctrls}^{bb}(\mathtt{a},\mathtt{c})$. ◁

Function 2: Adorn. For each adorned predicate $\mathtt{p}^\alpha$ produced by DMS, the function **Adorn** propagates the binding information of $\mathtt{p}^\alpha$ into all rules defining $\mathtt{p}$. In this step, we have to take into account the peculiarities of DLP$^{\mathcal{A}_m,\neg_s}$ programs, in order to define a suitable notion of SIPS to propagate binding information in presence of aggregate operators. Our strategy is defined next.

Definition 1 (SIPS). A SIPS for a DLP$^{\mathcal{A}_m,\neg_s}$ rule r w.r.t. a binding α for an atom $\mathtt{p}(\bar{\mathtt{t}}) \in H(r)$ is a pair $(\prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}, f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})})$, where:

– $\prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}$ is a strict partial order over the atoms in ATOMS(r); $\prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}$ is such that $\mathtt{p}(\bar{\mathtt{t}}) \prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})} \mathtt{q}(\bar{\mathtt{s}})$ holds for all atoms $\mathtt{q}(\bar{\mathtt{s}}) \in$ ATOMS(r) different from $\mathtt{p}(\bar{\mathtt{t}})$, and $\mathtt{q_i}(\bar{\mathtt{s}}_\mathtt{i}) \prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})} \mathtt{q_j}(\bar{\mathtt{s}}_\mathtt{j})$ implies that atom $\mathtt{q_i}(\bar{\mathtt{s}}_\mathtt{i})$ belongs to $B^+(r) \cup \{\mathtt{p}(\bar{\mathtt{t}})\}$;
– $f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}$ is a function assigning to each atom $\mathtt{q}(\bar{\mathtt{s}}) \in$ ATOMS(r) a subset of the variables in $\bar{\mathtt{s}}$ — intuitively, those made bound after processing $\mathtt{q}(\bar{\mathtt{s}})$; $f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}$ is such that $f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}(\mathtt{p}(\bar{\mathtt{t}}))$ contains all and only the variables of $\mathtt{p}(\bar{\mathtt{t}})$ corresponding to bound arguments in $\mathtt{p}^\alpha$. □

The propagation is performed according to the above strategy. In particular, for a binding α associated with an atom $\mathtt{p}(\bar{\mathtt{t}})$ in the head of a rule r, the SIPS $(\prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}, f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})})$ determines which variables are bound in the evaluation of each atom of r: A variable $\mathtt{X}$ of an atom $\mathtt{q}(\bar{\mathtt{s}})$ in ATOMS(r) is bound if and only if either (i) $\mathtt{X} \in f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}(\mathtt{p}(\bar{\mathtt{t}}))$ or (ii) there exists $\mathtt{b}(\bar{\mathtt{v}}) \in B^+(r)$ such that $\mathtt{b}(\bar{\mathtt{v}}) \prec_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})} \mathtt{q}(\bar{\mathtt{s}})$ and $\mathtt{X} \in f_r^{\mathtt{p}^\alpha(\bar{\mathtt{t}})}(\mathtt{b}(\bar{\mathtt{v}}))$.

Example 3. **Adorn** is invoked for the rule r_3 and the adorned atom $\mathtt{ctrls}^{bb}(\mathtt{X},\mathtt{Y})$. Let us assume that the adopted SIPS is as follows:

– $\mathtt{ctrls(X,Y)} \prec_{r_3}^{\mathtt{ctrls}^{bb}(\mathtt{X,Y})} \mathtt{comp(X)}$; $\quad \mathtt{ctrls(X,Y)} \prec_{r_3}^{\mathtt{ctrls}^{bb}(\mathtt{X,Y})} \mathtt{comp(Y)}$;
– $\mathtt{ctrls(X,Y)} \prec_{r_3}^{\mathtt{ctrls}^{bb}(\mathtt{X,Y})} \mathtt{cs(X,Z,Y,S)}$;
– $f_{r_3}^{\mathtt{ctrls}^{bb}(\mathtt{X,Y})}(\mathtt{ctrls(X,Y)}) = \{\mathtt{X,Y}\}$; $\quad f_{r_3}^{\mathtt{ctrls}^{bb}(\mathtt{X,Y})}(\mathtt{comp(X)}) = \{\mathtt{X}\}$;
– $f_{r_3}^{\mathtt{ctrls}^{bb}(\mathtt{X,Y})}(\mathtt{comp(Y)}) = \{\mathtt{Y}\}$; $\quad f_{r_3}^{\mathtt{ctrls}^{bb}(\mathtt{X,Y})}(\mathtt{cs(X,Z,Y,S)}) = \{\mathtt{X,Y}\}$.

According to the above SIPS, the following adorned rule is generated:

$$r_3^a : \ \mathtt{ctrls}^{bb}(\mathtt{X,Y}) \ \text{:--} \ \mathtt{comp(X)},\ \mathtt{comp(Y)},\ \#\mathtt{sum}\{\mathtt{S,Z} : \ \mathtt{cs}^{bfbf}(\mathtt{X,Z,Y,S})\} > 50.$$

Note that only IDB predicates are adorned in the rule above. A new adorned predicate, $\mathtt{cs}^{bfbf}$, has been produced, which will be propagated in a subsequent invocation of **Adorn**, for instance according to the following SIPS for r_1 and r_2:

- $\mathtt{cs(X,Z,Y,S)} \prec_{r_1}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}} \mathtt{owns(X,Y,S)} \prec_{r_1}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}} \mathtt{Z = X}$;
- $f_{r_1}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}}(\mathtt{cs(X,Z,Y,S)}) = f_{r_1}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}}(\mathtt{owns(X,Y,S)}) = \{\mathtt{X,Y}\}$;
- $f_{r_1}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}}(\mathtt{Z = X}) = \{\mathtt{X,Z}\}$;
- $\mathtt{cs(X,Z,Y,S)} \prec_{r_2}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}} \mathtt{owns(Z,Y,S)} \prec_{r_2}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}} \mathtt{ctrls(X,Z)}$;
- $f_{r_2}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}}(\mathtt{cs(X,Z,Y,S)}) = \{\mathtt{X,Y}\}$;
- $f_{r_2}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}}(\mathtt{owns(Z,Y,S)}) = f_{r_2}^{\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}}(\mathtt{ctrls(Z,Y)}) = \{\mathtt{Z,Y}\}$.

Hence, the following adorned rules will be produced:

$$\begin{aligned} r_1^a:&\ \ \mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}\ \text{:–}\ \mathtt{owns(X,Y,S)},\ \mathtt{Z = X}.\\ r_2^a:&\ \ \mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}\ \text{:–}\ \mathtt{owns(Z,Y,S)},\ \mathtt{ctrls}^{bb}\mathtt{(X,Z)}. \end{aligned}$$ ◁

Function 3: Generate. When **Generate** is invoked for an adorned rule r^a, which has been obtained by adorning a rule r w.r.t. an adorned head atom $\mathtt{p}^{\alpha}(\bar{\mathtt{t}})$, a magic rule r^* is produced for each $\mathtt{p_i}^{\alpha_i}(\bar{\mathtt{t}}_\mathtt{i}) \in \mathrm{ATOMS}(r^a)$ different from $\mathtt{p}^{\alpha}(\bar{\mathtt{t}})$ and such that $\mathtt{p_i}$ is an IDB predicate: The head atom of r^* is $\mathtt{mgc}(\mathtt{q_i}^{\beta_i}(\bar{\mathtt{s}}_\mathtt{i}))$, and the body of r^* consists of $\mathtt{mgc}(\mathtt{p}^{\alpha}(\bar{\mathtt{t}}))$ and all atoms $\mathtt{q_j}^{\beta_j}(\bar{\mathtt{s}}_\mathtt{j})$ in $B^+(r)$ such that $\mathtt{q_j}(\bar{\mathtt{s}}_\mathtt{j}) \prec_r^{\alpha} \mathtt{q_i}(\bar{\mathtt{s}}_\mathtt{i})$ holds.

Example 4. When **Generate** is invoked for r_3^a and $\mathtt{ctrls}^{bb}\mathtt{(X,Y)}$, the following magic rule is produced:

$$r_3^*:\ \ \mathtt{mgc\_cs}^{bfbf}\mathtt{(X,Y)}\ \text{:–}\ \mathtt{mgc\_ctrls}^{bb}\mathtt{(X,Y)}.$$

When **Generate** is invoked for r_1^a and $\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}$, no magic rules are generated because only EDB predicates appear in the body of r_1^a. Finally, when **Generate** is invoked for r_2^a and $\mathtt{cs}^{bfbf}\mathtt{(X,Z,Y,S)}$, the following magic rule is produced:

$$r_2^*:\ \ \mathtt{mgc\_ctrls}^{bb}\mathtt{(X,Z)}\ \text{:–}\ \mathtt{mgc\_cs}^{bfbf}\mathtt{(X,Y)},\ \mathtt{owns(Z,Y,S)}.$$ ◁

Function 4: Modify. Given an adorned rule r^a, obtained from a rule r, **Modify** builds and returns a modified rule r'. The modified rule r' is obtained from r by adding to its body a magic atom $\mathtt{mgc}(\mathtt{p}^{\alpha}(\bar{\mathtt{t}}))$ for each adorned atom $\mathtt{p}^{\alpha}(\bar{\mathtt{t}})$ occurring in $H(r^a)$. These magic atoms limit the range of the head variables during program instantiation.

Example 5. The modified rules in $\mathrm{DMS}(\mathcal{Q}_1, \mathcal{P}_1)$ are:

$$\begin{aligned} r_1':&\ \ \mathtt{cs(X,Z,Y,S)}\ \text{:–}\ \mathtt{mgc\_cs}^{bfbf}\mathtt{(X,Y)},\ \mathtt{owns(X,Y,S)},\ \mathtt{Z = X}.\\ r_2':&\ \ \mathtt{cs(X,Z,Y,S)}\ \text{:–}\ \mathtt{mgc\_cs}^{bfbf}\mathtt{(X,Y)},\ \mathtt{owns(Z,Y,S)},\ \mathtt{ctrls(X,Z)}.\\ r_3':&\ \ \mathtt{ctrls(X,Y)}\ \text{:–}\ \mathtt{mgc\_ctrls}^{bb}\mathtt{(X,Y)},\ \mathtt{comp(X)},\ \mathtt{comp(Y)},\\ &\qquad\qquad \#\mathtt{sum}\{\mathtt{S,Z:\ cs(X,Z,Y,S)}\} > 50. \end{aligned}$$

To sum up, the complete program $\mathrm{DMS}(\mathcal{Q}_1, \mathcal{P}_1)$ comprises the modified rules above, the query seed $r_{\mathcal{Q}_1}$, and the magic rules r_3^*, r_2^*. Note that $\mathcal{Q}_1$ is a brave and cautious consequence of $\mathcal{P}_1$ and of $\mathrm{DMS}(\mathcal{Q}_1, \mathcal{P}_1)$. ◁

4 Query Equivalence Theorem

In this section, we show the correctness of DMS for $\mathrm{DLP}^{\mathcal{A}_m,\neg_s}$ programs. The proof uses a suitable extension of the notion of unfounded set for $\mathrm{DLP}^{\mathcal{A}_m,\neg_s}$ programs. We will use the notion of unfounded set introduced in [27], opportunely adapted to our notation, and a theorem which is implicit in that paper.

Definition 2 (Unfounded Set). *Let $\langle T, U\rangle$ be a partial interpretation for a* $\mathrm{DLP}^{\mathcal{A}_m,\neg_s}$ *program $\mathcal{P}$ and $X \subseteq \mathcal{B}_\mathcal{P}$ be a set of atoms. Then, X is an* unfounded set *for $\mathcal{P}$ w.r.t. $\langle T, U\rangle$ if and only if, for each ground rule $r \in Ground(\mathcal{P})$ with $X \cap H(r) \neq \emptyset$, at least one of the following conditions holds: (i) $B^-(r) \cap T \neq \emptyset$; (ii) $B^+(r) \not\subseteq U$; (iii) $B^+(r) \cap X \neq \emptyset$; (iv) some (monotone) aggregate literal in $B^\mathcal{A}(r)$ is false w.r.t. $\langle T \setminus X, U \setminus X\rangle$; (v) $H(r) \cap (T \setminus X) \neq \emptyset$.*

Theorem 1. *Let $\langle T, U\rangle$ be a partial interpretation for a* $\mathrm{DLP}^{\mathcal{A}_m,\neg_s}$ *program $\mathcal{P}$. Then, for any stable model M of $\mathcal{P}$ such that $T \subseteq M \subseteq U$, and for each unfounded set X of $\mathcal{P}$ w.r.t. $\langle T, U\rangle$, $M \cap X = \emptyset$ holds.*

Proof. From Proposition 7 of [27], M is unfounded–free, that is, M is disjoint from every unfounded set for $\mathcal{P}$ w.r.t. M. Since X is an unfounded set for $\mathcal{P}$ w.r.t. M because of Proposition 1 of [27], $M \cap X = \emptyset$ follows. □

We will provide a link between unfounded sets and magic atoms by means of the following set of "killed" atoms.

Definition 3 (Killed Atoms). *Let $\mathcal{Q}$ be a query over a* $\mathrm{DLP}^{\mathcal{A}_m,\neg_s}$ *program $\mathcal{P}$, M' a model of* $\mathrm{DMS}(\mathcal{Q},\mathcal{P})$*, and $N' \subseteq M'$ a model of $Ground(\mathrm{DMS}(\mathcal{Q},\mathcal{P}))^{M'}$. The set $\mathtt{killed}^{M'}_{\mathcal{Q},\mathcal{P}}(N')$ of the* killed atoms *for* $\mathrm{DMS}(\mathcal{Q},\mathcal{P})$ *w.r.t. M' and N' is defined as $\{\mathtt{p}(\bar{\mathtt{t}}) \in \mathcal{B}_\mathcal{P} \setminus N' \mid$ either $\mathtt{p}$ is an EDB predicate or there is a binding α such that $\mathtt{mgc}(\mathtt{p}^\alpha(\bar{\mathtt{t}})) \in M'\}$.*

Intuitively, killed atoms are either false ground instances of some EDB predicate or false atoms which are relevant w.r.t. $\mathcal{Q}$ (they have associated magic atoms in the model N'). In terms of a hypothetical top–down evaluation of $\mathcal{Q}$, this means that killed atoms would be considered as subqueries but discovered to be false.

Theorem 2. *Let $\mathcal{Q}$ be a query over a* $\mathrm{DLP}^{\mathcal{A}_m,\neg_s}$ *program $\mathcal{P}$. Let M' be a model for* $\mathrm{DMS}(\mathcal{Q},\mathcal{P})$*, $N' \subseteq M'$ a model of $Ground(\mathrm{DMS}(\mathcal{Q},\mathcal{P}))^{M'}$. Then, $\mathtt{killed}^{M'}_{\mathcal{Q},\mathcal{P}}(N')$ is an unfounded set for $\mathcal{P}$ w.r.t. $\langle M' \cap \mathcal{B}_\mathcal{P}, \mathcal{B}_\mathcal{P}\rangle$.*

Proof (Sketch). Let $X = \mathtt{killed}^{M'}_{\mathcal{Q},\mathcal{P}}(N')$. Let $r' \in \mathrm{DMS}(\mathcal{Q},\mathcal{P})$ be a rule associated with a rule $r \in \mathcal{P}$ such that $H(r)\vartheta \cap \mathtt{killed}^{M'}_{\mathcal{Q},\mathcal{P}}(N') \neq \emptyset$ (ϑ a substitution). Since M' is a model of $\mathrm{DMS}(\mathcal{Q},\mathcal{P})$, M' satisfies $r'\vartheta$, i.e., at least one of the following conditions holds: (1) $B^-(r')\vartheta \cap M' \neq \emptyset$; (2) $B^+(r')\vartheta \not\subseteq M'$; (3) an aggregate $A \in B^\mathcal{A}(r')\vartheta$ is false w.r.t. M'; (4) $H(r')\vartheta \cap M' \neq \emptyset$. If (1) or (2) hold, we can show that either (i) or (iii) in Definition 2 hold. If (3) holds, by assuming $B^+(r')\vartheta \subseteq M'$ (i.e., (2) does not hold), we can observe that all (ground) standard atoms in the aggregate set of A are either in M' or in X. Thus, A is false w.r.t. $(M' \cap \mathcal{B}_\mathcal{P}) \setminus X$ and, in particular, A is false w.r.t. $\langle (M' \cap \mathcal{B}_\mathcal{P}) \setminus X, \mathcal{B}_\mathcal{P} \setminus X \rangle$, i.e., condition (iv) of Definition 2 holds. Finally, if (4) holds, by assuming that all previous cases do not hold, we can show that either (iii) or (v) hold. □

We are then ready to prove the soundness of stable model correspondence for DMS for $\text{DLP}^{\mathcal{A}_m,\neg_s}$ programs.

Theorem 3 (Soundness). *Let $\mathcal{Q}$ be a query over a $\text{DLP}^{\mathcal{A}_m,\neg_s}$ program $\mathcal{P}$. Then, for each stable model M' of $\text{DMS}(\mathcal{Q},\mathcal{P})$ and for each substitution ϑ, there is a stable model M of $\mathcal{P}$ such that $\mathcal{Q}\vartheta \in M$ if and only if $\mathcal{Q}\vartheta \in M'$.*

Proof (Sketch). By using Theorem 2, we can show that each stable model M of $\mathcal{P} \cup (M' \cap \mathcal{B}_\mathcal{P})$, the program obtained by adding to $\mathcal{P}$ a fact for each atom in $M' \cap \mathcal{B}_\mathcal{P}$, is in turn a stable model of $\mathcal{P}$ containing $M' \cap \mathcal{B}_\mathcal{P}$. Thus, we have that $\mathcal{Q}\vartheta \in M' \Longrightarrow \mathcal{Q}\vartheta \in M$ holds. Moreover, we can show $(*)$ $\mathcal{Q}\vartheta \notin M' \Longrightarrow \mathcal{Q}\vartheta \notin M$. To this aim, we note that $\texttt{killed}^{M'}_{\mathcal{Q},\mathcal{P}}(M')$ contains all instances of $\mathcal{Q}$ which are false w.r.t. M' (the magic seed is associated with each instance of $\mathcal{Q}$ in $\mathcal{B}_\mathcal{P}$) and is an unfounded set for $\mathcal{P}$ w.r.t. $\langle M' \cap \mathcal{B}_\mathcal{P}, \mathcal{B}_\mathcal{P} \rangle$ by Theorem 2. Therefore, $(*)$ follows from Theorem 1. □

For proving the completeness of stable model correspondence for DMS, we construct an interpretation for $\text{DMS}(\mathcal{Q},\mathcal{P})$ based on one for $\mathcal{P}$. The new interpretation is referred to as "magic variant" and is defined below.

Definition 4 (Magic Variant). *Consider a query $\mathcal{Q}$ over a $\text{DLP}^{\mathcal{A}_m,\neg_s}$ program $\mathcal{P}$. Let I be an interpretation for $\mathcal{P}$. We define an interpretation $\texttt{var}^{\infty}_{\mathcal{Q},\mathcal{P}}(I)$ for $\text{DMS}(\mathcal{Q},\mathcal{P})$, called the magic variant of I w.r.t. $\mathcal{Q}$ and $\mathcal{P}$, as the fixpoint of the following sequence:*

$$
\begin{aligned}
\texttt{var}^{0}_{\mathcal{Q},\mathcal{P}}(I) &= \text{EDB}(\mathcal{P}) \\
\texttt{var}^{i+1}_{\mathcal{Q},\mathcal{P}}(I) &= \texttt{var}^{i}_{\mathcal{Q},\mathcal{P}}(I) \cup \\
&\quad \{\texttt{p}(\bar{\texttt{t}}) \in I \mid \exists\, \alpha \textit{ such that } \texttt{mgc}(\texttt{p}^\alpha(\bar{\texttt{t}})) \in \texttt{var}^{i}_{\mathcal{Q},\mathcal{P}}(I)\} \cup \\
&\quad \{\texttt{mgc}(\texttt{p}^\alpha(\bar{\texttt{t}})) \mid \exists\, r^* \in \textit{Ground}(\text{DMS}(\mathcal{Q},\mathcal{P})) \textit{ such that} \\
&\qquad \texttt{mgc}(\texttt{p}^\alpha(\bar{\texttt{t}})) \in H(r^*) \textit{ and } B^+(r^*) \subseteq \texttt{var}^{i}_{\mathcal{Q},\mathcal{P}}(I)\}, \quad \forall i \geq 0.
\end{aligned}
$$

Theorem 4 (Completeness). *Let $\mathcal{Q}$ be a query over a $\text{DLP}^{\mathcal{A}_m,\neg_s}$ program $\mathcal{P}$. Then, for each stable model M of $\mathcal{P}$ and for each substitution ϑ, there is a stable model $M' = \texttt{var}^{\infty}_{\mathcal{Q},\mathcal{P}}(M)$ of $\text{DMS}(\mathcal{Q},\mathcal{P})$ such that $\mathcal{Q}\vartheta \in M$ if and only if $\mathcal{Q}\vartheta \in M'$.*

Proof (Sketch). By using Theorem 1 and the restriction to monotone aggregates, we can show that M' is a stable model of $\text{DMS}(\mathcal{Q},\mathcal{P})$ such that $M \supseteq M' \cap \mathcal{B}_\mathcal{P}$. Thus, we have that $\mathcal{Q}\vartheta \in M' \Longrightarrow \mathcal{Q}\vartheta \in M$ holds. Moreover, we can show $(*)$ $\mathcal{Q}\vartheta \notin M' \Longrightarrow \mathcal{Q}\vartheta \notin M$. To this aim, we note that $\texttt{killed}^{M'}_{\mathcal{Q},\mathcal{P}}(M')$ contains all instances of $\mathcal{Q}$ which are false w.r.t. M' (the magic seed is associated with each instance of $\mathcal{Q}$ in $\mathcal{B}_\mathcal{P}$) and is an unfounded set for $\mathcal{P}$ w.r.t. $\langle M' \cap \mathcal{B}_\mathcal{P}, \mathcal{B}_\mathcal{P} \rangle$ by Theorem 2. Therefore, from Theorem 1 we conclude $(*)$. □

Finally, we can prove the correctness of query answering for DMS for $\text{DLP}^{\mathcal{A}_m,\neg_s}$.

Theorem 5 (Query Equivalence Theorem). *Let $\mathcal{Q}$ be a query over a $\text{DLP}^{\mathcal{A}_m,\neg_s}$ program $\mathcal{P}$. Then, $\mathcal{P} \equiv^b_\mathcal{Q} \text{DMS}(\mathcal{Q},\mathcal{P})$ and $\mathcal{P} \equiv^c_\mathcal{Q} \text{DMS}(\mathcal{Q},\mathcal{P})$ hold.*

Proof. Since $\mathcal{P}$ is coherent (i.e., $\mathcal{P}$ admits at least one stable model) and the DMS algorithm does not depend on EDB facts, the theorem is a consequence of Theorem 3 and Theorem 4. □

5 Experimental Results

Benchmark Problems and Data. For assessing the efficacy of the approach, we designed three benchmark settings. First, we considered Company Controls. In our benchmark, in an instance with n companies there are $\delta_1 = n/3$ direct controls and $\delta_i = \delta_{i-1}/2$ controls at distance i, for each $i \geq 2$. Moreover, $n^{1.5}$ facts for `owns` were randomly introduced. For each instance size, we generated three instances, and for each instance we considered three different queries: $\texttt{ctrls(a,b)}$ (*bound–bound*), $\texttt{ctrls(a,Y)}$ (*bound–free*) and $\texttt{ctrls(X,b)}$ (*free–bound*). Companies `a` and `b` have maximal control distance. The tested encoding is reported in Fig. 2.

We also considered two benchmarks using data from the WordNet ontology. In the benchmarks, we considered queries asking whether a given word is a hyponym (resp. a hypernym) of k+ words, for some constant k. We recall that w is a hyponym of w' if the semantic field of w is included within that of w'; in this case, w' is a hypernym of w'. More specifically, for evaluating the scalability of DMS, the original dataset was multiplied by introducing arbitrary prefixes to ids and words. On these enlarged datasets, we considered three queries, each of which with a different value for k. The tested encoding, taken in part from the OpenRuleBench project, is reported below (for hypernym queries, $\texttt{W}$ and $\texttt{W}_1$ in the aggregate literal of the last rule are inverted):

$$\texttt{tc(X,Y)} \text{ :– } \texttt{hyp(X,Y)}. \quad \texttt{tc(X,Y)} \text{ :– } \texttt{tc(X,Z)},\ \texttt{hyp(Z,Y)}.$$
$$\texttt{hypernym}(\texttt{W}_1,\texttt{W}_2) \text{ :– } \texttt{s}(\texttt{ID}_1,\_,\_,\texttt{S}_1,\_,\_),\ \texttt{tc}(\texttt{ID}_1,\texttt{ID}_2),\ \texttt{s}(\texttt{ID}_2,\_,\texttt{W}_2,\_,\_,\_).$$
$$\texttt{hypernym}_{k+}(\texttt{W}) \text{ :– } \texttt{s}(\_,\_,\texttt{W},\_,\_,\_),\ \#\texttt{count}\{\texttt{W}_1 : \texttt{hypernym}(\texttt{W}_1,\texttt{W})\} \geq k.$$

Results and Discussion. The experiment were performed on a 3GHz Intel® Xeon® processor system with 4GB RAM under the Debian 4.0 operating system with a GNU/Linux 2.6.23 kernel. The tested DLV system was compiled with GCC 4.1.3. For

Table 1. Company Controls: Average execution time (seconds)

Number of	*bb* Queries		*bf* Queries		*fb* Queries	
Companies	No Magic	DMS	No Magic	DMS	No Magic	DMS
10 000	4.09	0.23	4.09	0.36	4.08	0.24
20 000	9.19	0.50	9.18	0.78	9.24	0.53
30 000	14.12	0.78	14.05	1.19	14.01	0.82
40 000	19.35	1.04	19.21	1.61	19.17	1.09
50 000	24.40	1.34	24.43	2.09	24.42	1.41
60 000	29.68	1.61	29.58	2.53	29.51	1.70
70 000	35.33	1.90	35.65	2.96	35.38	1.98
80 000	40.64	2.17	40.82	3.38	40.99	2.28
90 000	46.49	2.46	46.49	3.86	46.42	2.61
100 000	52.07	2.93	52.01	4.52	51.74	3.09

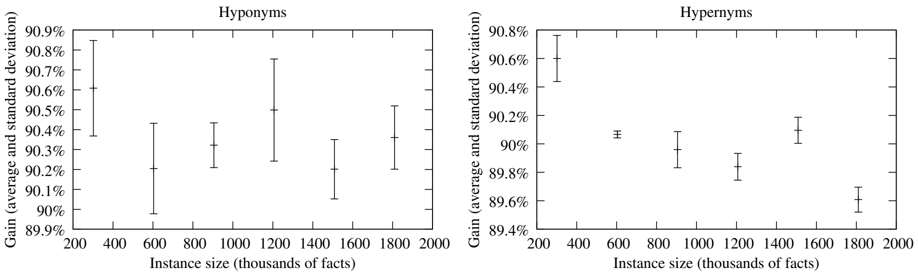


Fig. 3. Benchmarks on WordNet: Average gain and standard deviation

every instance, we allowed a maximum running time of 600 seconds (10 minutes) and a maximum memory usage of 3GB. Experimental results for Company Controls are summarized in Table 1. For each instance size, we computed the average execution time of DLV with and without DMS (we performed rewritings manually). The advantages of the Magic Set method clearly emerge. Concerning the benchmarks on WordNet, we report in Fig. 3 the average gain provided by DMS[8]. It can be observed that, in all considered queries, DMS provide a sensible performance gain (90% or more) to DLV.

6 Conclusion

Aggregate functions in logic programming languages appeared already in the 80s, when their need emerged in deductive databases like LDL. Their importance from a knowledge representation perspective is now widely recognized, since they can be simulated only by means of inefficient and unnatural encodings of the problems. However, very few efforts have been spent to develop effective optimization strategies for logical programs enriched with aggregate operators. In this paper, we have presented a technique for the optimization of (partially) bound queries that extends the Magic Set method to the DLP$^{\mathcal{A}_m,\neg_s}$ language. An avenue of further research is to integrate the algorithms presented here with current approaches to optimize DLP programs with unstratified negations [22]. Moreover, it would be interesting to assess whether Magic Sets can be extended even to programs where aggregate literals are not necessarily monotone.

References

1. Lobo, J., Minker, J., Rajasekar, A.: Foundations of Disjunctive Logic Programming. The MIT Press, Cambridge (1992)
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS 22(3), 364–418 (1997)
3. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with Answer Set Programming in the Gioia-Tauro Seaport. TPLP (2011) (to appear)
4. Manna, M., Ruffolo, M., Oro, E., Alviano, M., Leone, N.: The HiLeX System for Semantic Information Extraction. TLDKS(2011) (to appear)
5. Ricca, F., Alviano, M., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: A Logic–Based System for e-Tourism. FI 105, 35–55 (2010)

[8] Numerical details are reported in Appendix.

6. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL 7(3), 499–562 (2006)
7. Janhunen, T., Niemelä, I., Simons, P., You, J.H.: Partiality and Disjunctions in Stable Model Semantics. In: KR 2000, April 12-15, pp. 411–419 (2000)
8. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 447–451. Springer, Heidelberg (2005)
9. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: KR 2008, pp. 422–432. AAAI Press, Menlo Park (2008)
10. Kemp, D.B., Stuckey, P.J.: Semantics of Logic Programs with Aggregates. In: ISLP 1991, pp. 387–401. MIT Press, Cambridge (1991)
11. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Semantics for Logic Programs with Aggregates. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 212–226. Springer, Heidelberg (2001)
12. Dix, J., Osorio, M.: On Well-Behaved Semantics Suitable for Aggregation. In: ILPS 1997, Port Jefferson, N.Y (1997)
13. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI 138, 181–234 (2002)
14. Pelov, N., Truszczyński, M.: Semantics of disjunctive programs with monotone aggregates - an operator-based approach. In: NMR 2004, pp. 327–334 (2004)
15. Manna, M., Ricca, F., Terracina, G.: Consistent Query Answering via ASP from Different Perspectives: Theory and Practice. TPLP (2011) (to appear)
16. Ullman, J.D.: Principles of Database and Knowledge Base Systems, vol. 2. CS Press, Rockvillie (1989)
17. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: PODS 1986, pp. 1–16 (1986)
18. Beeri, C., Ramakrishnan, R.: On the power of magic. JLP 10(1-4), 255–259 (1991)
19. Kowalski, R.A.: Predicate Logic as Programming Language. In: IFIP Congress, pp. 569–574 (1974)
20. Greco, S.: Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. IEEE TKDE 15(2), 368–385 (2003)
21. Cumbo, C., Faber, W., Greco, G.: Improving Query Optimization for Disjunctive Datalog. In: APPIA-GULP-PRODE, pp. 252–262 (2003)
22. Faber, W., Greco, G., Leone, N.: Magic Sets and their Application to Data Integration. JCSS 73(4), 584–609 (2007)
23. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC 9, 365–385 (1991)
24. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
25. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
26. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the dlv system. TPLP 8(5-6), 545–580 (2008)
27. Alviano, M., Faber, W., Leone, N.: Using unfounded sets for computing answer sets of programs with recursive aggregates. In: CILC 2007 (2007)
28. Alviano, M.: Dynamic Magic Sets for Disjunctive Datalog Programs. In: ICLP 2010. LIPIcs, vol. 7, pp. 226–235 (2010)

Strong Equivalence of Logic Programs with Abstract Constraint Atoms

Guohua Liu[1], Randy Goebel[2], Tomi Janhunen[1], Ilkka Niemelä[1], and Jia-Huai You[2]

[1] Aalto University, Department of Information and Computer Science
{Guohua.Liu,Tomi.Janhunen,Ilkka.Niemela}@aalto.fi
[2] University of Alberta, Department of Computing Science
{goebel,you}@cs.ualberta.ca

Abstract. Logic programs with abstract constraint atoms provide a unifying framework for studying logic programs with various kinds of constraints. Establishing strong equivalence between logic programs is a key property for program maintenance and optimization, and for guaranteeing the same behavior for a revised original program in any context. In this paper, we study strong equivalence of logic programs with abstract constraint atoms. We first give a general characterization of strong equivalence based on a new definition of program reduct for logic programs with abstract constraints. Then we consider a particular kind of program revision—constraint replacements addressing the question: under what conditions can a constraint in a program be replaced by other constraints, so that the resulting program is strongly equivalent to the original one.

1 Introduction

Logic programming interpreted with *answer set* semantics or answer set programming (ASP), is a declarative programming paradigm for knowledge representation, designed for characterizing and solving computationally hard problems [1,2]. In ASP, a problem is represented by a logic program and the answer sets of the program correspond to the solutions to the problem. Answer set programming with *abstract constraint atoms* (*c-atoms*) [3,4,5] provides a unifying framework for the study of logic programs with various constraints, such as weight constraints [6], aggregates [7,8], and global constraints [9].

Strong equivalence within this kind of semantics [10] is one of the key concepts of logic programming. A program (a set of rules) P is *strongly equivalent* to a program Q if, for any other program R, the programs $P \cup R$ and $Q \cup R$ have the same answer sets. In order to see whether a set of rules in a program can always be replaced by another set of rules, regardless of other program components, one needs to check whether the two sets of rules are strongly equivalent. Strongly equivalent programs are guaranteed to have the same behavior in any context. Uniform equivalence [11] is a special case of strong equivalence. Uniformly equivalent programs have the same behavior in any context of facts. Lifschitz et. al. [10] developed a characterization of strong equivalence using the logic of here-and-there. Lin [12] presented a transformation by which the strong equivalence of logic programs is converted to classical entailment. Turner [13] provided a model-theoretical characterization of the strong equivalence, where two programs are strongly equivalent if and only if they have the same set of SE-models. Liu and Truszczyński [11] extended this approach to logic programs with monotone constraints.

In this paper, we study the characterization of strong equivalence for logic programs with arbitrary abstract constraint atoms, under the semantics based on conditional satisfaction [5]. We extend the concept of program *reduct* to logic programs with abstract constraint atoms. Using the notion of program reduct, we define SE-models and UE-models in the standard way employed in [11,13] and characterize strong and uniform equivalence by SE-models and UE-models, respectively. Then, we study strong equivalence of a particular class of program revisions, viz. *constraint replacements*, that amount to replacing a constraint in a program by another constraint or a combination of constraints. Constraint replacements can be used as program transformations to decompose a complicated constraint into simpler parts when doing program development or optimization. We note that constraint replacements are also a standard technique to implement complicated constraints in current ASP systems: typically the inference engine of the system supports a limited set of basic constraints and more involved constraints are compiled to basic ones during grounding [6,14]. Here, we are interested in replacements that can be applied in any context, i.e., where the original program and the modified program are strongly equivalent. Strong equivalence is particularly valuable because it allows replacements to be done in either the whole or a part of the program range, while (weak) equivalence only consider the former. We provide fundamental results on replacement operations by presenting criteria under which a constraint can be replaced with a conjunction, a disjunction, or a combination of constraints while preserving strong equivalence. An observation is that replacements with disjunctions are more involved and require an extra condition compared to those with conjunctions.

This paper is organized as follows. The next section reviews the basic definitions of programs with general abstract constraints under the semantics based on conditional satisfaction [5]. In Section 3, we characterize strong equivalence by the SE-models of constraint programs and show that the characterization generalizes to uniform equivalence. Section 4 applies strong equivalence in the study of constraint replacements. In Section 5, we address the interconnections of our results on constraint replacement to existing transformations. Related work is described in Section 6. Finally, we conclude the paper in Section 7.

2 Preliminaries

We review the answer set semantics for logic programs with arbitrary constraint atoms, as defined in [5]. The semantics should also be contributed to [15], where it is defined as a fix-point construction using 3-valued logic. We assume a propositional language with a countable set of propositional *atoms*.

An *abstract constraint atom* (*c-atom*) is a construct of the form (D, C) where D is the *domain* of the c-atom and C the *admissible solution set* of the c-atom. The domain D is a finite set of atoms and the admissible solution set C is a set of subsets of D, i.e., $C \subseteq 2^D$. Given a c-atom $A = (D, C)$, we use A_d and A_c to refer to sets D and C, respectively. Certain special c-atoms have been distinguished. A c-atom of the form $(\{a\}, \{\{a\}\})$ simply denotes a propositional atom a. A c-atom A is *monotone* if for every $X \subseteq Y \subseteq A_d$, $X \in A_c$ implies that $Y \in A_c$, *antimonotone* if for every $X \subseteq Y \subseteq A_d$, $Y \in A_c$ implies that $X \in A_c$, and *convex* if for every $X \subseteq Y \subseteq Z \subseteq A_d$, $X \in A_c$ and $Z \in A_c$ implies $Y \in A_c$.

A logic program with c-atoms, also called a *constraint program* (or *program* for short), is a finite set of rules of the form

$$A \leftarrow A_1, \ldots, A_n. \tag{1}$$

where A and A_i's are c-atoms.

For a program P, we denote by $At(P)$ the set of atoms appearing in P. In general, negative atoms of the form `not` A may appear in a rule. Following [5], a negative c-atom `not` A in a program is interpreted as, and substituted by, its *complement* c-atom $\overline{A}$, where $\overline{A}_d = A_d$ and $\overline{A}_c = 2^{A_d} \setminus A_c$. Due to this assumption, we consider the programs where no c-atoms appear negatively.

For a rule r of the form (1), we define $hd(r) = A$ and $bd(r) = \{A_1, ..., A_n\}$, which are called the *head* and the *body* of r, respectively. A rule r is said to be *basic* if $hd(r)$ is a propositional atom[1]. A program P is *basic* if every rule in it is basic and *normal* if every c-atom in it is a propositional atom.

A set of atoms M *satisfies* a c-atom A, written $M \models A$, if $M \cap A_d \in A_c$. Otherwise M does not satisfy A, written $M \not\models A$. Satisfaction naturally extends to conjunctions and disjunctions of c-atoms.

Answer sets for constraint programs are defined in two steps. First, answer sets for basic programs are defined, based on the notion *conditional satisfaction*. Then the answer sets for general programs are defined.

Definition 1. *Let S and M be sets of atoms such that $S \subseteq M$. The set S* conditionally satisfies *a c-atom A, w.r.t. M, denoted by $S \models_M A$, if $S \models A$ and for every $I \subseteq A_d$ such that $S \cap A_d \subseteq I$ and $I \subseteq M \cap A_d$, we have that $I \in A_c$.*

Example 1. Let A be the c-atom $(\{a, b\}, \{\emptyset, \{a\}, \{a, b\}\})$ and $S_1 = \emptyset$, $S_2 = \{a\}$, and $M = \{a, b\}$. Then, $S_1 \not\models_M A$ and $S_2 \models_M A$. $\square$

Conditional satisfaction extends naturally to conjunctions and disjunctions of c-atoms. Whenever it is clear by the context, we may use a set of c-atoms to denote a conjunction or a disjunction of c-atoms.

An operator T_P is defined as follows: for any sets S, M, and a basic program P, $T_P(S, M) = \{a \mid \exists r \in P,\ hd(r) = a, \text{ and } S \models_M bd(r)\}$. The operator T_P is monotonic w.r.t its first argument (given that the second argument is fixed). Answer sets of a basic program P are defined as the (least) fixpoint of T_P.

[1] The head can also be $\bot$, which denotes the c-atom $(D, \emptyset)$. Such a rule serves as a constraint [5]. Rules of this kind are irrelevant for the purposes of this paper.

Definition 2. *Let P be a basic program and M a set of atoms. The set M is an* answer set *of P iff M is a model of P and $M = T_P^\infty(\emptyset, M)$, where $T_P^0(\emptyset, M) = \emptyset$ and $T_P^{i+1}(\emptyset, M) = T_P(T_P^i(\emptyset, M), M)$, for all $i \geq 0$.*

The answer sets of a (general) program are defined on the basis of the answer sets of a basic program—the *instance* of the general program. Let P be a constraint program, r a rule in P of the form (1), and M a set of atoms. The instance of r, with respect to M, is

$$inst(r, M) = \begin{cases} \{a \leftarrow bd(r) \mid a \in M \cap hd(r)_d\}, & \text{if } M \models hd(r); \\ \emptyset, & \text{otherwise.} \end{cases}$$

The instance of P with respect to M, denoted $inst(P, M)$, is the program

$$inst(P, M) = \cup_{r \in P} inst(r, M)$$

Definition 3. *Let P be a program and M a set of atoms. The set M is an answer set of P iff M is an answer set of $inst(P, M)$.*

3 Characterization of Strong and Uniform Equivalence

We first define the reduct of c-atoms and general constraint programs. Then, using the reduct, we define SE-models and characterize the strong equivalence of programs. Finally, we show how these results extend to uniform equivalence.

3.1 Program Reduct

The program reduct plays a very central role in the definition of answer sets for normal programs [1]. However, it is non-trivial to generalize the reduct (e.g. [16]). In what follows, we propose a new way of reducing c-atoms themselves, establish a close connection between conditional satisfaction of c-atoms and satisfaction of reduced c-atoms, and then extend these ideas to cover rules and programs.

Definition 4. *Let A be a c-atom and M a set of atoms. The* reduct *of A, w.r.t. M, denoted A^M, is the c-atom (A_d^M, A_c^M), where $A_d^M = A_d$ and the set of admissible solutions $A_c^M = \{S \mid S \in A_c,\ S \subseteq M,\ \text{and } S \models_M A\}$.*

Proposition 1. *Let A be a c-atom and S and M be sets of atoms such that $S \subseteq M$. Then $S \models_M A$ iff $S \models A^M$.*

Example 2. Let $A = (\{a, b, c\}, \{\{a\}, \{a, b\}, \{a, c\}, \{a, b, c\}\})$ be a c-atom. Then, given an interpretation $M = \{a, b\}$, we have $A^M = (\{a, b, c\}, \{\{a\}, \{a, b\}\})$. □

Definition 5. *Let P be a basic program and M a set of atoms. The* reduct *of P, w.r.t. M, denoted P^M, is the program obtained by:*

1. *removing from P any rules whose bodies are not satisfied by M;*
2. *replacing each c-atom with the reduct of the c-atom w.r.t. M, in the bodies of the remaining rules.*

Definition 6. *Let P be a program and M a set of atoms. The reduct of P, w.r.t M, denoted P^M, is the reduct of the instance of P w.r.t. M, i.e., $inst(P, M)^M$.*

3.2 Strong Equivalence

Strong equivalence can be defined in the standard way using the notion of answer sets from Definition 3, independently of the class of programs. Similarly, given Definition 6, the notion of SE-models can be adopted—paving the way for Theorem 1 which characterizes strong equivalence in terms of SE-models.

Definition 7. *Programs P and Q are* strongly equivalent, *denoted $P \equiv_s Q$, iff, for any program R, the programs $P \cup R$ and $Q \cup R$ have the same answer sets.*

Definition 8. *Let P be a program. A pair of sets (X, Y) is a* strong equivalence model (SE-model) *of P if the following conditions hold: (1) $X \subseteq Y$; (2) $Y \models P$; and (3) $X \models P^Y$. The set of SE-models of P is denoted by* $\mathrm{SE}(P)$.

Theorem 1. *Let P and Q be two programs. Then, $P \equiv_s Q$ iff* $\mathrm{SE}(P) = \mathrm{SE}(Q)$.

Proof Sketch. We use $\mathrm{AS}(P)$ to denote the set of answer sets of a program P.

($\Longrightarrow$) Let (X, Y) be an SE-model of P. We show that (X, Y) is also an SE-model of Q, by contradiction. Assume that $Y \not\models Q$. Consider the program $R = \{a \mid a \in Y\}$. We can show that $Y \in \mathrm{AS}(P \cup R)$ and $Y \notin \mathrm{AS}(Q \cup R)$, contradicting $P \equiv_s Q$. Assume $X \not\models Q^Y$. Consider the program $R = \{a \mid a \in X\} \cup \{b \leftarrow c \mid b \in Y \text{ and } c \in S \setminus X\}$ where $S = \{a \mid \text{there is } r \in Q^Y \text{ such that } hd(r) = a \text{ and } X \models bd(r)\}$. We can show that $Y \notin \mathrm{AS}(P \cup R)$ and $Y \in \mathrm{AS}(Q \cup R)$, contradicting $P \equiv_s Q$. So, $(X, Y) \in \mathrm{SE}(Q)$. It follows by symmetry that any SE-model of Q is also an SE-model of P. Therefore $\mathrm{SE}(P) = \mathrm{SE}(Q)$.

($\Longleftarrow$) It is easy to show the following statement: for any programs P and Q, $\mathrm{SE}(P \cup Q) = \mathrm{SE}(P) \cap \mathrm{SE}(Q)$, and if $\mathrm{SE}(P) = \mathrm{SE}(Q)$, then $\mathrm{AS}(P) = \mathrm{AS}(Q)$. So, given $\mathrm{SE}(P) = \mathrm{SE}(Q)$, we have for all programs R, $\mathrm{SE}(P \cup R) = \mathrm{SE}(Q \cup R)$ and $\mathrm{AS}(P \cup R) = \mathrm{AS}(Q \cup R)$. Therefore $P \equiv_s Q$. □

3.3 Uniform Equivalence

The concept of uniform equivalence is closely related to strong equivalence (cf. Definition 7). The essential difference is that in uniform equivalence, the context program R is restricted to be a set of facts. To formalize this, we use a special rule $r_D = (D, \{D\}) \leftarrow$ for any set of atoms D [11]. Adding the rule r_D to a program is then equivalent to adding each atom $a \in D$ as a fact $(\{a\}, \{\{a\}\}) \leftarrow$.

Definition 9. *Programs P and Q are* uniformly equivalent, *denoted $P \equiv_u Q$, iff, for any set of atoms D, $P \cup \{r_D\}$ and $Q \cup \{r_D\}$ have the same answer sets.*

The uniform equivalence of finite programs can be characterized similarly as that in [11] using *uniform equivalence models* which are a special class of SE-models. We consider finite programs only as the uniform equivalence of infinite programs cannot be captured by any class of SE-models in general [17].

Definition 10. *Let P be a program. A pair (X, Y) is a* uniform equivalence model (UE-model) *of P if the following conditions hold: (1) (X, Y) is an SE-model of P; (2) for every SE-model (X', Y) of P such that $X \subseteq X'$, either $X' = X$ or $X' = Y$. The set of UE-models of P is denoted by* $\mathrm{UE}(P)$.

Theorem 2. *For any finite programs P and Q, $P \equiv_u Q$ iff* $\mathrm{UE}(P) = \mathrm{UE}(Q)$.

4 Constraint Replacements

In this section we consider a particular kind of program revision—*constraint replacement*. The idea is that a constraint represented by a c-atom in a logic program is replaced by either (i) a conjunction of constraints, (ii) a disjunction of constraints, or (iii) a combination of them. It is then natural to use strong equivalence as correctness criterion and we establish explicit conditions on which strong equivalence is preserved. As regards notation, we define the *Cartesian* product of two sets of interpretations S_1 and S_2, denoted $S_1 \times S_2$, as the set of interpretations $\{T_1 \cup T_2 \mid T_1 \in S_1 \text{ and } T_2 \in S_2\}$. Using this notion, we are able to define a basic operation for constructing sets of admissible solutions.

Definition 11. *The* extension of the set A_c of admissible solutions *of a c-atom A over a set D of atoms, denoted by $ext(A_c, D)$, is $ext(A_c, D) = A_c \times 2^{(D \setminus A_d)}$.*

Proposition 2. *For a c-atom A, a set D of atoms, and an interpretation M, the extended projection $M \cap (A_d \cup D) \in ext(A_c, D)$ iff $M \models A$.*

Given a rule r of the form (1) and $A_k \in bd(r)$ with $1 \leq k \leq n$, we write $r[A_k/B_1, \ldots, B_m]$ for the result of substituting c-atoms $B_1, \ldots, B_m$ for A_k, i.e.,

$$A \leftarrow A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n. \tag{2}$$

4.1 Conjunctive Encoding

In a conjunctive encoding, the idea is to represent a c-atom A as a conjunction of c-atoms $A_1, \ldots, A_m$ where each A_i may have a subdomain of A_d as its domain.

Definition 12. *A conjunction of c-atoms $A_1, \ldots, A_m$ is a* conjunctive encoding *of a c-atom A, denoted $A = \mathcal{C}(A_1, \ldots, A_m)$, iff the c-atoms satisfy*

1. $A_d = \bigcup_{i=1}^{m} (A_i)_d$; *and*
2. $A_c = \bigcap_{i=1}^{m} ext((A_i)_c, A_d)$.

The conditions of Definition 12 guarantee important properties for conjunctive encodings as detailed below: The (conditional) satisfaction of c-atoms is preserved and the same can be observed for the reducts of c-atoms.

Proposition 3. *If $A = \mathcal{C}(A_1, \ldots, A_m)$, then for any M, N such that $M \subseteq N$:*

1. $M \models A$ *iff* $M \models A_i$ *for each* $1 \leq i \leq m$;
2. $M \models_N A$ *iff* $M \models_N A_i$ *for each* $1 \leq i \leq m$; *and*
3. $M \models A^N$ *iff* $M \models (A_i)^N$ *for each* $1 \leq i \leq m$.

The properties listed above guarantee that replacing a c-atom in a program by its conjunctive encoding $A_1, \ldots, A_m$ also preserves SE-models. This observation leads us to the following results, at both the rule and program levels.

Theorem 3. *Let r be a rule of the form (1) and A_k a c-atom in the body $bd(r)$. If $A_k = \mathcal{C}(B_1, \ldots, B_m)$, then $\{r\} \equiv_s \{r[A_k/B_1, \ldots, B_m]\}$.*

In the above, the rule $r[A_k/B_1, \ldots, B_m]$ coincides with (2) and we call this particular rule the *conjunctive rewrite* of r with respect to $A_k = \mathcal{C}(B_1, \ldots, B_m)$. Since $\equiv_s$ is a congruence relation, i.e., $P \equiv_s Q$ implies $P \cup R \equiv_s Q \cup R$, we can apply Theorem 3 in any context. In particular, we call a program P' a *one-step conjunctive rewrite* of P iff P' is obtained as $(P \setminus \{r\}) \cup \{r[A_k/B_1, \ldots, B_m]\}$ for $A_k \in bd(r)$ and $A_k = \mathcal{C}(B_1, \ldots, B_m)$. This idea easily generalizes for n steps.

Corollary 1. *For an n-step conjunctive rewrite P' of a program P, $P \equiv_s P'$.*

It is also worth pointing out special cases of conjunctive encodings. If each domain $(A_i)_d$ coincides with A_d, then $ext((A_i)_c, A_d) = (A_i)_c$ and the second condition of Definition 12 implies $A_c = \bigcap_{i=1}^{m} (A_i)_c$. On the other hand, if the domains of each A_i and A_j with $i \neq j$ are mutually disjoint, then the second condition reduces to a Cartesian product $A_c = (A_1)_c \times \ldots \times (A_m)_c$. In other intermediate cases, we obtain a *natural join* $A_c = (A_1)_c \bowtie \ldots \bowtie (A_m)_c$ condition, which was introduced by Janhunen et al. [18] to relate the set of answer sets associated with an entire logic program with those of its component programs.

4.2 Disjunctive Encoding

The idea of a disjunctive encoding is to represent a c-atom A as a disjunction of c-atoms $A_1, \ldots, A_m$. However, in contrast with Definition 12, an additional condition becomes necessary in order to preserve conditional satisfaction of c-atoms and their reducts.

Definition 13. *A disjunction of c-atoms $A_1, \ldots, A_m$ is a* disjunctive encoding *of a c-atom A, denoted $A = \mathcal{D}(A_1, \ldots, A_m)$, iff the c-atoms satisfy*

1. *$A_d = \bigcup_{i=1}^{m} (A_i)_d$;*
2. *$A_c = \bigcup_{i=1}^{m} ext((A_i)_c, A_d)$; and*
3. *for any subset M of A_d, $A_c^M = \bigcup_{i=1}^{m} (A_i)_c^M$.*

Proposition 4. *If $A = \mathcal{D}(A_1, \ldots, A_m)$, then for any M, N such that $M \subseteq N$:*

1. *$M \models A$ iff $M \models A_i$ for some $1 \leq i \leq m$;*
2. *$M \models_N A$ iff $M \models_N A_i$ for some $1 \leq i \leq m$; and*
3. *$M \models A^N$ iff $M \models (A_i)^N$ for some $1 \leq i \leq m$.*

Because of the general properties of disjunction, we need to be careful about disjunctive encodings when replacing c-atoms in rules. Rewriting a rule r of the form (1) with respect to a disjunctive encoding $A_k = \mathcal{D}(B_1, \ldots, B_m)$ results in m rules $r[A_k/B_1], \ldots, r[A_k/B_m]$ obtained by substituting A_k by each B_i in turn. Proposition 4 guarantees the preservation of strong equivalence.

Theorem 4. *Let r be a rule of the form (1) and A_k a c-atom in the body $bd(r)$. If $A_k = \mathcal{D}(B_1, \ldots, B_m)$, then $\{r\} \equiv_s \{r[A_k/B_1], \ldots, r[A_k/B_m]\}$.*

Hence, in *one-step disjunctive rewriting* based on Theorem 4, a program P with $r \in P$ would be rewritten as $P' = (P \setminus \{r\}) \cup \{r[A_k/B_1], \ldots, r[A_k/B_m]\}$. This also preserves strong equivalence by Theorem 4. In general, we obtain:

Corollary 2. *For an n-step disjunctive rewrite P' of a program P, $P \equiv_s P'$.*

The condition 3 of Definition 13 reveals that, in contrast with conjunctive encodings, conditional satisfaction is not automatically preserved in the disjunctive case. The next example illustrates that the first two conditions that preserve the satisfaction of a c-atom are insufficient to preserve strong equivalence.

Example 3. Let P be the following program with an *aggregate* denoted by A:

$$p(2) \leftarrow A. \qquad p(1) \leftarrow . \qquad p(-3) \leftarrow p(2).$$

The intuitive reading of A is $\mathsf{SUM}(\{X \mid p(X)\}) \neq -1$ and following [5], it corresponds to a c-atom with $A_d = \{p(1), p(3), p(-3)\}$ and $A_c = 2^{A_d} \setminus \{\{p(2), p(-3)\}\}$. It may seem natural to replace A by the disjunction of $A_1 = \mathsf{SUM}(\{X \mid p(X)\}) > -1$ and $A_2 = \mathsf{SUM}(\{X \mid p(X)\}) < -1$ and, therefore, to rewrite P as P':

$$p(2) \leftarrow A_1. \qquad p(2) \leftarrow A_2. \qquad p(1) \leftarrow . \qquad p(-3) \leftarrow p(2).$$

However, the programs P and P' are not strongly equivalent. To check this, consider $M = \{p(1), p(2), p(-3)\}$ which is an answer set of P but not that of P'. This aspect is captured by the third condition of Definition 13 because A_c^M differs from $(A_1)_c^M \cup (A_2)_c^M$ for the interpretation $M = \{p(1), p(2), p(-3)\}$. □

There is also one interesting special case of disjunctive encodings conforming to Definition 13. If the domain of each c-atom A_i coincides with A_d, i.e., $(A_i)_d = A_d$ for each $1 \leq i \leq m$, then $ext((A_i)_c, A_d) = (A_i)_c$ for each $1 \leq i \leq m$ as well. Thus, the sets of admissible solutions are simply related by $A_c = \bigcup_{i=1}^{m} (A_i)_c$.

4.3 Shannon Encodings

Any Boolean function $f(a_1, \ldots, a_n)$ can be expanded with respect to its argument a_i using Shannon's partial evaluation principle:

$$f(a_1, \ldots, a_n) = (a_i \wedge f(a_1, \ldots, a_{i-1}, \top, a_{i+1}, \ldots, a_n)) \vee \\ (\neg a_i \wedge f(a_1, \ldots, a_{i-1}, \bot, a_{i+1}, \ldots, a_n)). \quad (3)$$

The objective of this section is to present Shannon expansion for *monotone* c-atoms. The reason for this restriction is that Shannon's principle cannot be applied to arbitrary c-atoms in a natural way (see Example 5 for details). In the case of monotone c-atoms, however, the following can be established.

Proposition 5. *If A is a monotone c-atom and $a \in A_d$, then it holds that $A = \mathcal{D}(\mathcal{C}(a, A^+(a)), A^-(a))$ where a stands for the c-atom $(\{a\}, \{\{a\}\})$, and*

1. *$A^+(a) = (A_d \setminus \{a\}, \{T \setminus \{a\} \mid T \in A_c$ and $a \in T\})$ and*
2. *$A^-(a) = (A_d \setminus \{a\}, \{T \mid T \in A_c$ and $a \notin T\})$.*

Given this relationship we may call $\mathcal{S}(A,a) = \mathcal{D}(\mathcal{C}(a, A^+(a)), A^-(a))$ as the *Shannon encoding* of A with respect to an atom $a \in A_d$. Intuitively, the Shannon encoding $\mathcal{S}(A,a)$ builds on a case analysis. The part $\mathcal{C}(a, A^+(a))$ captures admissible solutions of A where a is true. The part $A^-(a)$ covers cases where a is false (by default) and hence $(\{a\}, \{\emptyset\})$ is not incorporated. We call $A^+(a)$ and $A^-(a)$ the respective *positive* and *negative* encodings of A given $a \in A_d$.

Example 4. Consider a monotone c-atom $A = (\{a,b\}, \{\{a\}, \{a,b\}\})$ for which $\mathcal{S}(A,a) = \mathcal{D}(\mathcal{C}(a, A^+(a)), A^-(a))$ where the respective positive and negative encodings of A are $A^+(a) = (\{b\}, \{\emptyset, \{b\}\})$ and $A^-(a) = (\{b\}, \{\})$. It is worth noting that the latter c-atom is never satisfied and if it is used to rewrite any rule body, the resulting rule can be directly omitted due to inapplicability. □

Given a monotone c-atom A, any atom $a \in A_d$ can be used to do the Shannon encoding. When the identity of a is not important, we simply use $\mathcal{S}(A, \cdot)$ to denote the appropriate construction, the properties of which are as follows.

Proposition 6. *If A is a monotone c-atom, then so are $A^+(\cdot)$ and $A^-(\cdot)$.*

Proposition 7. *If A is a monotone c-atom and $a \in A_d$, then for any M, N such that $M \subseteq N$:*

1. *$M \models A$ iff $M \models a \wedge A^+(a)$ or $M \models A^-(a)$;*
2. *$M \models_N A$ iff $M \models_N a \wedge A^+(a)$ or $M \models_N A^-(a)$; and*
3. *$M \models A^N$ iff $M \models a \wedge A^+(a)^N$ or $M \models A^-(a)^N$.*

We stress that the Shannon encoding $\mathcal{S}(A,a)$ is not even satisfaction preserving if applied to other than monotone c-atoms. This is illustrated below.

Example 5. Consider the antimonotone c-atom $A = (\{a\}, \{\emptyset\})$. We have that $A^+(a) = (\emptyset, \emptyset)$, $A^-(a) = (\emptyset, \{\emptyset\})$, and $\mathcal{S}(A,a) = \mathcal{D}(\mathcal{C}(a, A^+(a)), A^-(a)) \equiv (\emptyset, \{\emptyset\})$. Let $M = \{a\}$. It is easy to see that $M \models \mathcal{S}(A,a)$. But, on the other hand, we have $M \not\models A$. □

However, for monotone c-atoms, strong equivalence is additionally preserved under Shannon encodings. Since $\mathcal{S}(A,a)$ is a combination of disjunctive and conjunctive encodings, our preceding results on rewriting rules and programs apply. Given a rule r of the form (1), a monotone c-atom A_k in the body $bd(r)$, and an atom $a \in (A_k)_d$, the *Shannon rewrite* of r consists of two rules $r[A_k/(\{a\}, \{\{a\}\}), A^+(a)]$ and $r[A_k/A^-(a)]$. Such replacements are highly beneficial if either $A^+(a)$ or $A^-(a)$ becomes trivial in one sense (cf. Example 4). If not, then repeated Shannon rewritings can lead to an exponential expansion.

Theorem 5. *Let r be a rule of the form (1), A_k a monotone c-atom in the body $bd(r)$, and $a \in (A_k)_d$ an atom. Then $\{r\} \equiv_s \{r[A_k/a, A^+(a)], r[A_k/A^-(a)]\}$ where $A^+(a)$ and $A^-(a)$ are the respective positive and negative encodings of A.*

Corollary 3. *For an n-step Shannon rewrite P' of a program P, $P \equiv_s P'$.*

It is also possible to rewrite a program P by mixing conjunctive, disjunctive, and Shannon rewriting (Shannon rewriting can be only done for monotone c-atoms). Corollaries 1, 2, and 3 guarantee, on their behalf, that the resulting program will be strongly equivalent with the original one.

5 Interconnections to Some Existing Encodings

In this section, we work out the interconnections of some existing translations of c-atoms in the literature to conjunctive and disjunctive encodings. In this way, we can establish that these transformations preserve strong equivalence by appealing to the results of Section 4.

Liu and Truszczyński [11] propose a way of representing any convex c-atom A as a conjunction of two c-atoms A^+ and A^- that are the *upward* and *downward* closures of A, respectively. These closures are defined by

1. $A^+_d = A^-_d = A_d$,
2. $A^+_c = \{T \subseteq A_d \mid S \subseteq T \text{ for some } S \in A_c\}$, and
3. $A^-_c = \{T \subseteq A_d \mid T \subseteq S \text{ for some } S \in A_c\}$.

It is obvious that A^+ is monotone and A^- is antimonotone. In addition to this, the two conditions from Definition 12 can be verified so that $A = \mathcal{C}(A^+, A^-)$ holds in general. This justifies the statement of Proposition 8 given below. Thus it follows by Theorem 3 and Corollary 1 that when a convex c-atom A appearing in a program is replaced by its upward and downward closures A^+ and A^-, the resulting program is strongly equivalent to the original one.

Proposition 8. *The encoding of convex c-atoms in terms of their upward and downward closures [11] is a conjunctive encoding.*

As regards arbitrary c-atoms, a number of representations have been proposed such as *sampler sets* [16], the *translation* of aggregates into propositional formulas [19,20], *abstract representations* [21], and *local power sets* [22]. Given a c-atom A, these approaches are essentially based on a disjunctive encoding $A = \mathcal{D}(A_1, \ldots, A_m)$ where each A_i is defined by

1. $(A_i)_d = A_d$ and
2. $(A_i)_c = \{T \subseteq A_d \mid L_i \subseteq T \subseteq G_i\}$

where L_i and G_i are *least* and *greatest* admissible solutions from A_c such that (i) $L_i \subseteq G_i$, (ii) each T between L_i and G_i also belongs to A_c and (iii) the range induced by L_i and G_i is maximal in this sense. Intuitively, the sets L_i and $A_d \setminus G_i$ consist of atoms that have to be true and false, respectively, in order to satisfy A_i as well as A. It is obvious that each A_i is convex. For this reason we call an encoding of this kind as the *convex* encoding of A. Proposition 9 below is a consequence of verifying that A and $A_1, \ldots, A_m$ meet the requirements of Definition 13. So, referring to Theorem 4, given a rule r with an arbitrary c-atom in its body and a convex encoding $A = \mathcal{D}(A, \ldots, A_m)$ for A, the replacement of r by $r[A/A_1], \ldots, r[A/A_m]$ leads to a strongly equivalent program.

Proposition 9. *The convex encoding based on the representations of c-atoms in [16,19,20,21,22] is a disjunctive encoding.*

6 Related Work

The characterizations of strong equivalence in [10,12,11,13] provide the basis to determine the strong equivalence of normal programs under the answer set semantics [1]. For constraint programs under the answer set semantics, strong equivalence can be determined by translating the constraint programs to normal programs, then applying the previous characterizations. Given a constraint program, note that the resulting normal program may be exponential in the size of the original program [20,8,23]. The characterization of strong equivalence developed in this paper makes it possible to determine whether strong equivalence holds without such a potentially exponential translation.

Example 6. Let P be a program consisting of the rule: $b \leftarrow A$, where A is the aggregate $\mathsf{COUNT}(\{X \mid p(X)\}) \leq k$ and $p(X)$ is defined over the domain $D = \{p(1), ..., p(n)\}$.

Let N be a set of atoms such that $|N \cap D| \leq k$ and $b \in N$. Let M be any subset of N with $b \notin M$. It can be determined, without enumerating the admissible solutions of A, that $M \not\models P^N$, since $M \models_N A$ and $b \notin M$. So, (M, N) does not satisfy the third condition in the definition of SE-models. Then $(M, N) \notin \mathrm{SE}(P)$. So, for any program Q that has (M, N) as its SE-model, we can conclude that $P \not\equiv_s Q$.

On the other hand, to determine the strong equivalence of P and Q using the characterizations of strong equivalence for normal programs, one has to translate P to a normal program whose size is possibly exponential in the size of P (a straightforward translation consists of $\mathcal{O}\binom{n}{k}$ rules). □

The concept of program reduct (Definition 5) developed in this paper leads to a simple definition of answer sets for (disjunctive) constraint programs, where a rule head could be a disjunction of c-atoms. Given a (disjunctive) constraint program P and a set of atoms M, M is an answer set of P if and only if M is a minimal model of the program reduct P^M. It can be verified that the program reduct and the answer sets coincide with the generalized Gelfond-Lifschitz transformation and answer sets defined in [21]. Note, however, the simplicity of our definition.

Program revisions under strong and uniform equivalence have been studied in [24] for disjunctive logic programs. In that research, the concept of revision is motivated to remove redundant rules in a program. In contrast, we study a wider class of programs with constraint atoms and the question how such constraint atoms can be replaced by others, independently of the embedding program.

7 Conclusion and Directions of Future Work

We propose the concept of reduct for logic programs with arbitrary abstract constraint atoms (c-atoms), based on which we give a characterization of strong equivalence and uniform equivalence of logic programs with c-atoms. This extends previous work on logic programs with monotone c-atoms [11]. We study strong equivalence of a particular class of program revisions: constraint replacements. We provide criteria under which a c-atom can be replaced by a conjunction of c-atoms, a disjunction c-atoms, or a combination of them while preserving strong equivalence. These results provide the basis to check if a constraint can be replaced with other constraints in any program context.

For the future work, it would be interesting to extend our results to programs with concrete constraints such as aggregate programs [7,25]. Another promising direction is that to investigate the strong equivalence of logic programs embedded within other reasoning mechanisms, such as constraint programming (CP) [26,27,28], description logics [29], and SAT modulo theories [30]. These embeddings make it possible to exploit the strengths of other reasoning mechanisms in ASP for solving complex real world problems. In the embedding approach, it is a common practice to replace a part of a program with components of other reasoning mechanisms to model and solve different kinds of problems. The study of strong equivalence in this context may provide insights and approaches to program optimization, system implementation, and efficient answer set computation.

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. ICLP, pp. 1070–1080 (1988)
2. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Math. and Artificial Intelligence 25(3-4), 241–273 (1999)
3. Marek, V., Niemelä, I., Truszczyński, M.: Logic programs with monotone abstract constraint atoms. TPLP 8(2), 167–199 (2008)
4. Marek, V.W., Remmel, J.B.: Set constraints in logic programming. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 167–179. Springer, Heidelberg (2003)
5. Son, T.C., Pontelli, E., Tu, P.H.: Answer sets for logic programs with arbitrary abstract constraint atoms. JAIR 29, 353–389 (2007)
6. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
7. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
8. Son, T.C., Pontelli, E.: A constructive semantic characterization of aggregates in answer set programming. TPLP 7, 355–375 (2006)
9. van Hoeve, W.J., Katriel, I.: Global constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
10. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2(4), 526–541 (2001)
11. Liu, L., Truszczyński, M.: Properties and applications of programs with monotone and convex constraints. JAIR 7, 299–334 (2006)
12. Lin, F.: Reducing strong equivalence of logic programs to entailment in classical propositional logic. In: Proc. KR 2002, pp. 170–176 (2002)
13. Turner, H.: Strong equivalence made easy: nested expressions and weight constraints. Theory and Practice of Logic Programming 3, 609–622 (2003)

14. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder *gringo*. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 502–508. Springer, Heidelberg (2009)
15. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. TPLP 7(3), 301–353 (2007)
16. Janhunen, T.: Sampler programs: The stable model semantics of abstract constraint programs revisited. In: Proc. ICLP, pp. 94–103 (2010)
17. Eiter, T., Fink, M., Woltran, S.: Semantical characterizations and complexity of equivalences in answer set programming. ACM Transactions on Computational Logic 8(3) (2007)
18. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. J. Artif. Intell. Res. (JAIR) 35, 813–857 (2009)
19. Pelov, N.: Semantics of Logic Programs with Aggregates. PhD thesis, Ketholieke Universiteit Leuven (2004)
20. Pelov, N., Denecker, M., Bruynooghe, M.: Translation of aggregate programs to normal logic programs. In: Proc. ASP 2003, pp. 29–42 (2003)
21. Shen, Y., You, J., Yuan, L.: Characterizations of stable model semantics for logic programs with arbitrary constraint atoms. TPLP 9(4), 529–564 (2009)
22. You, J., Liu, G.: Loop formulas for logic programs with arbitrary constraint atoms. In: Proc. AAAI 2008, pp. 584–589 (2008)
23. You, J., Yuan, L.Y., Liu, G., Shen, Y.: Logic programs with abstract constraints: Representation, disjunction and complexities. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 228–240. Springer, Heidelberg (2007)
24. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 87–99. Springer, Heidelberg (2003)
25. Ferraris, P.: Answer sets for propositional theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 119–131. Springer, Heidelberg (2005)
26. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
27. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 235–249. Springer, Heidelberg (2009)
28. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Math. and AI 53(1-4), 251–287 (2008)
29. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artificial. Intelligence. 172(12-13), 1495–1539 (2008)
30. Niemelä, I.: Integrating answer set programming and satisfiability modulo theories. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, p. 3. Springer, Heidelberg (2009)

Back and Forth between Rules and SE-Models[⋆]

Martin Slota and João Leite

CENTRIA & Departamento de Informática
Universidade Nova de Lisboa
Quinta da Torre
2829-516 Caparica, Portugal

Abstract. Rules in logic programming encode information about mutual interdependencies between literals that is not captured by any of the commonly used semantics. This information becomes essential as soon as a program needs to be modified or further manipulated.

We argue that, in these cases, a program should not be viewed solely as the set of its models. Instead, it should be viewed and manipulated as the *set of sets of models* of each rule inside it. With this in mind, we investigate and highlight relations between the SE-model semantics and individual rules. We identify a set of representatives of rule equivalence classes induced by SE-models, and so pinpoint the exact expressivity of this semantics with respect to a single rule. We also characterise the class of sets of SE-interpretations representable by a single rule. Finally, we discuss the introduction of two notions of equivalence, both stronger than *strong equivalence* [1] and weaker than *strong update equivalence* [2], which seem more suitable whenever the dependency information found in rules is of interest.

1 Motivation

In this paper we take a closer look at the relationship between the SE-model semantics and individual rules of a logic program. We identify a set of representatives of rule equivalence classes, which we dub *canonical rules*, characterise the class of sets of SE-interpretations that are representable by a single rule, and show how the corresponding canonical rules can be reconstructed from them. We believe that these results pave the way to view and manipulate a logic program as the *set of sets of SE-models* of each rule inside it. This is important in situations when the set of SE-models of the whole program fails to capture essential information encoded in individual rules inside it, such as when the program needs to be modified or further manipulated. With this in mind, we briefly discuss two new notions of equivalence, stronger than *strong equivalence* [1] and weaker than *strong update equivalence* [2].

In many extensions of Answer-Set Programming, individual rules of a program are treated as first-class citizens – apart from their prime role of encoding the answer sets assigned to the program, they carry essential information about mutual interdependencies between literals that cannot be captured by answer sets. Examples that enjoy these

[⋆] An extended version of this paper with all the proofs is available at `http://arxiv.org/abs/1102.5385` characteristics include the numerous approaches that deal with dynamics of logic programs, where inconsistencies between older and newer knowledge need to be resolved by "sacrificing" parts of an older program (such as in [3,4,5,6,7,8,9,10,11]). These approaches look at subsets of logic programs in search of plausible conflict resolutions. Some of them go even further and consider particular literals in heads and bodies of rules in order to identify conflicts and find ways to solve them. This often leads to definitions of new notions which are *too* syntax-dependent. At the same time, however, semantic properties of the very same notions need to be analysed, and their syntactic basis then frequently turns into a serious impediment.

Arguably, a *more* syntax-independent method for this kind of operations would be desirable. Not only would it be theoretically more appealing, but it would also allow for a better understanding of its properties with respect to the underlying semantics. Moreover, such a more semantic approach could facilitate the establishment of bridges with the area of Belief Change (see [12] for an introduction), and benefit from the many years of research where semantic change operations on monotonic logics have been studied, desirable properties for such operations have been identified, and constructive definitions of operators satisfying these properties have been introduced.

However, as has repeatedly been argued in the literature [4,13], fully semantic methods do not seem to be appropriate for the task at hand. Though their definition and analysis is technically possible and leads to very elegant and seemingly desirable properties, there are a number of simple examples for which these methods fail to provide results that would be in line with basic intuitions [4]. Also, as shown in [13], these individual problems follow a certain pattern: intuitively, any purely semantic approach to logic program updates satisfying a few very straightforward and desirable properties cannot comply with the property of *support* [14,15], which lies at the very heart of semantics for Logic Programs. This can be demonstrated on simple programs $\mathcal{P} = \{ p., q. \}$ and $\mathcal{Q} = \{ p., q \leftarrow p. \}$ which are *strongly equivalent*, thus indistinguishable from the semantic perspective, but while $\mathcal{P}$ does not contain any dependencies, $\mathcal{Q}$ introduces a dependence of atom q upon atom p. This has far-reaching consequences, at least with respect to important notions from the logic programming point of view, such as that of *support*, which are themselves defined in syntactic rather than semantic terms. For example, if we change our beliefs about p, and come to believe that it is false, we may expect different beliefs regarding q, depending on whether we start form $\mathcal{P}$, in which case q would still be true, or $\mathcal{Q}$, in which case q would no longer be true because it is no longer supported.

We believe that rules indeed contain information that, to the best of our knowledge, cannot be captured by any of the existing semantics for Logic Programs. In many situations, this information is essential for making further decisions down the line. Therefore, any operation on logic programs that is expected to respect syntax-based properties like *support* cannot operate solely on the semantic level, but rather has to look inside the program and acknowledge rules as the atomic pieces of knowledge. At the same time, however, rules need not be manipulated in their original form. The abstraction provided by Logic Programming semantics such as SE-models can be used to discard unimportant differences between the syntactic forms of rules and focus on their semantic content. Thus, while a program cannot be viewed as the set of its models for reasons described above, it can still be viewed as a *set of sets of models* of rules in it. Such a shift of focus should make the approach easier to manage theoretically, while not neglecting the importance of literal dependencies expressed in individual rules. It could also become a bridge between existing approaches to rule evolution and properties as well as operator constructions known from Belief Change, not only highlighting the differences between them, but also clarifying why such differences arise in the first place.

However, before a deeper investigation of such an approach can begin, we do need to know more about the relation of SE-models and individual rules. This is the aim of this paper, where we:

- identify a set of representatives of rule equivalence classes induced by the SE-model semantics, which we dub *canonical rules*;
- show how to reconstruct *canonical rules* from their sets of SE-models;
- based on the above, characterise the sets of SE-interpretations that are representable by a single rule;
- reveal connections between the set of SE-models of a rule and convex sublattices of the set of classical interpretations;
- introduce two new notions of equivalence – stronger than *strong equivalence* [1] and weaker than *strong update equivalence* [2] – and argue that they are more suitable when rules are to be treated as first-class citizens.

We believe that these results provide important insights into the workings of SE-models with respect to individual rules and will serve as a toolset for manipulating logic programs at the semantic level.

The rest of this document is structured as follows: We introduce syntax and semantics of logic programs in Sect. 2 while in Sect. 3 we define the set of representatives for rule equivalence classes and introduce transformations pinpointing the expressivity of SE-model semantics with respect to individual rules. We also give two characterisations of the sets of SE-interpretations that are representable by a single rule. In Sect. 4 we discuss the relevance of our results and propose the two new notions of equivalence.

2 Preliminaries

We assume to be given a nonempty, finite set of propositional atoms $\mathcal{L}$ from which we construct both propositional formulae and rules.

Propositional formulae are formed in the usual way from propositional atoms in $\mathcal{L}$, the logical constants $\top$ an $\bot$, and the connectives $\neg, \wedge, \vee, \subset, \supset, \equiv$. An *interpretation* is any subset of $\mathcal{L}$, naturally inducing a truth assignment to all propositional formulae. If a formula ϕ is true under interpretation I, we also say that I is a *model of* ϕ. The set of all interpretations is denoted by $\mathcal{I}$.

Similarly as for propositional formulae, the basic syntactic building blocks of rules are propositional atoms from $\mathcal{L}$. A *negative literal* is an atom preceded by $\sim$, denoting default negation. A *literal* is either an atom or a negative literal. As a convention, double default negation is absorbed, so that $\sim\sim p$ denotes the atom p. Given a set of literals X, we introduce the following notation:

$$X^+ = \{ p \in \mathcal{L} \mid p \in X \} \quad X^- = \{ p \in \mathcal{L} \mid \sim p \in X \} \quad \sim X = \{ \sim p \mid p \in X \cap \mathcal{L} \}$$

Given natural numbers k, l, m, n and atoms $p_1, \ldots, p_k$, $q_1, \ldots, q_l$, $r_1, \ldots, r_m$, $s_1, \ldots, s_n$, we say the pair of sets of literals

$$\langle \{ p_1, \ldots, p_k, \sim q_1, \ldots, \sim q_l \}, \{ r_1, \ldots, r_m, \sim s_1, \ldots, \sim s_n \} \rangle \tag{1}$$

is a *rule*. The first component of a rule (1) is denoted by $H(r)$ and the second by $B(r)$. We say $H(r)$ is the *head of* r, $H(r)^+$ is the *positive head of* r, $H(r)^-$ is the *negative head of* r, $B(r)$ is the *body of* r, $B(r)^+$ is the *positive body of* r and $B(r)^-$ is the *negative body of* r. Usually, for convenience, instead of a rule r of the form (1) we write the expression

$$p_1; \ldots; p_k; \sim q_1; \ldots; \sim q_l \leftarrow r_1, \ldots, r_m, \sim s_1, \ldots, \sim s_n. \tag{2}$$

or, alternatively, $H(r)^+; \sim H(r)^- \leftarrow B(r)^+, \sim B(r)^-$. A rule is called *positive* if its head and body contain only atoms. A *program* is any set of rules.

We also introduce the following non-standard notion which we will need throughout the rest of the paper:

Definition 1 (Canonical Tautology). *Let p_ε be an arbitrary but fixed atom. The* canonical tautology*, denoted by ε, is the rule $p_\varepsilon \leftarrow p_\varepsilon$.*

In the following, we define two semantics for rules. One is that of classical models, where a rule is simply treated as a classical implication. The other is based on the logic of Here-and-There [16,17], more accurately on a reformulation of the here-and-there semantics, called *SE-model semantics*, defined for rules [18]. This second semantics is strictly more expressive than both classical models and the stable model semantics [19].

We introduce the classical model of a rule by translating the rule into a propositional formula: Given a rule r of the form (2), we define the propositional formula $\overline{r}$ as $\bigvee \{ p_1, \ldots, p_k, \neg q_1, \ldots, \neg q_l \} \subset \bigwedge \{ r_1, \ldots, r_m, \neg s_1, \ldots, \neg s_n \}$. Note that $\bigvee \emptyset \equiv \bot$ and $\bigwedge \emptyset \equiv \top$. A classical model, or *C-model*, of a rule r is any model of the formula $\overline{r}$.

Given a rule r and an interpretation J, we define the *reduct of r relative to J*, denoted by r^J, as follows: If some atom from $H(r)^-$ is false under J or some atom from $B(r)^-$ is true under J, then r^J is ε; otherwise r^J is $H(r)^+ \leftarrow B(r)^+$. Intuitively, the reduct r^J is the positive part of a rule r that "remains" after all its negative literals are interpreted under interpretation J. The two conditions in the definition check whether the rule is satisfied based on the negative atoms in its head and body, interpreted under J. If this is the case, the reduct is by definition the canonical tautology. If none of these conditions is satisfied, the positive parts of r are kept in the reduct, discarding the negative ones.

An *SE-interpretation* is a pair of interpretations $\langle I, J \rangle$ such that I is a subset of J. The set of all SE-interpretations is denoted by $\mathcal{I}^{\mathsf{SE}}$. We say that an SE-interpretation $\langle I, J \rangle$ is an *SE-model* of a rule r if J is a C-model of r and I is a C-model of r^J. The set of all SE-models of a rule r is denoted by $\mathrm{mod}_{\mathsf{SE}}(r)$. The SE-models of a program $\mathcal{P}$ are the SE-models of all rules in $\mathcal{P}$. A set of SE-interpretations $\mathcal{S}$ is called **rule-representable** if there exists a rule r such that $\mathcal{S} = \mathrm{mod}_{\mathsf{SE}}(r)$.

We say that a rule r is **SE-tautological** if $\mathrm{mod}_{\mathsf{SE}}(r) = \mathcal{I}^{\mathsf{SE}}$. Note that the canonical tautology ε (c.f. Definition 1) is SE-tautological. We say that two rules r, r' are *strongly equivalent*, or *SE-equivalent*, if they have the same set of SE-models.

3 Rule Equivalence Classes and Their Canonical Rules

Our goal is to find useful insights into the inner workings of the SE-model semantics with respect to single rules. In order to do so, we first introduce a set of representatives of rule equivalence classes induced by SE-models and show how the representative of a class can be constructed given one of its members. Then we show how to reconstruct a representative from the set of its SE-models. Finally, we pinpoint the conditions under which a set of SE-interpretations is rule-representable.

3.1 Canonical Rules

We start by bringing out simple but powerful transformations that simplify a given rule while preserving its SE-models. Most of these results have already been formulated in various ways [20,2,21]. The following result summarises the conditions under which a rule is SE-tautological:

Lemma 2 (Consequence of Theorem 4.4 in [2]; part i) of Lemma 2 in [21]). *Let H and B be sets of literals and p be an atom. Then a rule is SE-tautological if it takes any of the following forms:*

$$p; H \leftarrow p, B. \qquad H; {\sim}p \leftarrow B, {\sim}p. \qquad H \leftarrow B, p, {\sim}p.$$

Thus, repeating an atom in different "components" of the rule frequently causes the rule to be SE-tautological. In particular, this happens if the same atom occurs in the positive head and positive body, or in the negative head and negative body, or in the positive and negative bodies of a rule. How about the cases when the head contains a negation of a literal from the body? The following Lemma clarifies this situation:

Lemma 3 (Consequence of (3) and (4) in Lemma 1 in [21]). *Let H and B be sets of literals and L be a literal. Then rules of the following forms are SE-equivalent:*

$$H; {\sim}L \leftarrow L, B. \qquad H \leftarrow L, B. \tag{3}$$

So if a literal is present in the body of a rule, its negation can be removed from the head.

Until now we have seen that a rule r that has a common atom in at least two of the sets $H(r)^+ \cup H(r)^-$, $B(r)^+$ and $B(r)^-$ is either SE-tautological, or SE-equivalent to a rule where the atom is omitted from the rule's head. So such a rule is always SE-equivalent either to the canonical tautology ε, or to a rule without such repetitions. Perhaps surprisingly, repetitions in positive and negative head cannot be simplified away. For example, over the alphabet $\mathcal{L}_p = \{ p \}$, the rule "$p; {\sim}p \leftarrow .$" has two SE-models, $\langle \emptyset, \emptyset \rangle$ and $\langle \{ p \}, \{ p \} \rangle$, so it is not SE-tautological, nor is it SE-equivalent to any of the facts "$p.$" and "${\sim}p.$". Actually, it is not very difficult to see that it is not SE-equivalent to *any* other rule, even over larger alphabets. So the fact that an atom is in both $H(r)^+$ and $H(r)^-$ cannot all by itself imply that some kind of SE-models preserving rule simplification is possible.

The final Lemma reveals a special case in which we can eliminate the whole negative head of a rule and move it to its positive body. This occurs whenever the positive head is empty.

Lemma 4 (Related to Corollary 4.10 in [20] and Corollary 1 in [21]). *Let H^- be a set of negative literals, B be a set of literals and p be an atom. Then rules of the following forms are* SE*-equivalent:*

$$\sim p; H^- \leftarrow B. \qquad\qquad H^- \leftarrow p, B.$$

Armed with the above results, we can introduce the notion of a canonical rule. Each such rule represents a different equivalence class on the set of all rules induced by the SE-model semantics. In other words, every rule is SE-equivalent to exactly one canonical rule. After the definition, we provide constructive transformations which show that this is indeed the case. Note that the definition can be derived directly from the Lemmas above:

Definition 5 (Canonical Rule). *We say a rule r is* canonical *if either it is ε, or the following conditions are satisfied:*

1. *The sets $H(r)^+ \cup H(r)^-$, $B(r)^+$ and $B(r)^-$ are pairwise disjoint.*
2. *If $H(r)^+$ is empty, then $H(r)^-$ is also empty.*

This definition is closely related with the notion of a *fundamental rule* introduced in Definition 1 of [21]. There are two differences between canonical and fundamental rules: (1) a fundamental rule must satisfy condition 1. above, but need not satisfy condition 2.; (2) no SE-tautological rule is fundamental. As a consequence, fundamental rules do not cover all rule-representable sets of SE-interpretations, and two distinct fundamental rules may still be SE-equivalent. From the point of view of rule equivalence classes induced by SE-model semantics, there is one class that contains no fundamental rule, and some classes contain more than one fundamental rule. In the following we show that canonical rules overcome both of these limitations of fundamental rules. In other words, every rule is SE-equivalent to exactly one canonical rule. To this end, we define constructive transformations that directly show the mutual relations between rule syntax and semantics.

The following transformation provides a direct way of constructing a canonical rule that is SE-equivalent to a given rule r.

Definition 6 (Transformation into a Canonical Rule). *Given a rule r, by $\mathsf{can}(r)$ we denote a canonical rule constructed as follows: If any of the sets $H(r)^+ \cap B(r)^+$, $H(r)^- \cap B(r)^-$ and $B(r)^+ \cap B(r)^-$ is nonempty, then $\mathsf{can}(r)$ is ε. Otherwise, $\mathsf{can}(r)$ is of the form $H^+; \sim H^- \leftarrow B^+, \sim B^-$. where*

- *$H^+ = H(r)^+ \setminus B(r)^-$.*
- *If H^+ is empty, then $H^- = \emptyset$ and $B^+ = B(r)^+ \cup H(r)^-$.*
- *If H^+ is nonempty, then $H^- = H(r)^- \setminus B(r)^+$ and $B^+ = B(r)^+$.*
- *$B^- = B(r)^-$.*

Correctness of the transformation follows directly from Lemmas 2 to 4.

Theorem 7. *Every rule r is* SE*-equivalent to the canonical rule $\mathsf{can}(r)$.*

What remains to be proven is that no two different canonical rules are SE-equivalent. In the next Subsection we show how every canonical rule can be reconstructed from the set of its SE-models. As a consequence, no two different canonical rules can have the same set of SE-models.

3.2 Reconstructing Rules

In order to reconstruct a rule r from the set $\mathcal{S}$ of its SE-models, we need to understand how exactly each literal in the rule influences its models. The following Lemma provides a useful characterisation of the set of countermodels of a rule in terms of syntax:

Lemma 8 (Different formulation of Theorem 4 in [21]). *Let r be a rule. An SE-interpretation $\langle I, J\rangle$ is not an SE-model of r if and only if the following conditions are satisfied:*

1. *$H(r)^- \cup B(r)^+ \subseteq J$ and $J \subseteq \mathcal{L} \setminus B(r)^-$.*
2. *Either $J \subseteq \mathcal{L} \setminus H(r)^+$ or both $B(r)^+ \subseteq I$ and $I \subseteq \mathcal{L} \setminus H(r)^+$.*

The first condition together with the first disjunct of the second condition hold if and only if J is not a C-model of r. The second disjunct then captures the case when I is not a C-model of r^J.

If we take a closer look at these conditions, we find that the presence of a negative body atom in J guarantees that the first condition is falsified, so $\langle I, J\rangle$ is a model of r, regardless of the content of I. Somewhat similar is the situation with positive head atoms – whenever such an atom is present in I, it is also present in J, so the second condition is falsified and $\langle I, J\rangle$ is a model of r. Thus, if $\mathcal{S}$ is the set of SE-models of a rule r, then every atom $p \in B(r)^-$ satisfies

$$p \in J \text{ implies } \langle I, J\rangle \in \mathcal{S} \qquad (C_{B^-})$$

and every atom $p \in H(r)^+$ satisfies

$$p \in I \text{ implies } \langle I, J\rangle \in \mathcal{S} \ . \qquad (C_{H^+})$$

If we restrict ourselves to canonical rules different from ε, we find that these conditions are not only necessary, but, when combined properly, also sufficient to decide what atoms belong to the negative body and positive head of the rule.

For the rest of this Subsection, we assume that r is a canonical rule different from ε and $\mathcal{S}$ is the set of SE-models of r. Keeping in mind that every atom that satisfies condition (C_{B^-}) also satisfies condition (C_{H^+}) (because I is a subset of J), and that $B(r)^-$ is by definition disjoint from $H(r)^+$, we arrive at the following results:

Lemma 9. *An atom p belongs to $B(r)^-$ if and only if for all $\langle I, J\rangle \in \mathcal{I}^{\mathsf{SE}}$, the condition ($C_{B^-}$) is satisfied. An atom p belongs to $H(r)^+$ if and only if it does not belong to $B(r)^-$ and for all $\langle I, J\rangle \in \mathcal{I}^{\mathsf{SE}}$, the condition ($C_{H^+}$) is satisfied.*

As can be seen from Lemma 8, the role of positive body and negative head atoms is dual to that of negative body and positive head atoms. Intuitively, their absence in J, and sometimes also in I, implies that $\langle I, J\rangle$ is an SE-model of r. It follows from the first condition of Lemma 8 that if p belongs to $H(r)^- \cup B(r)^+$, then the following condition is satisfied:

$$p \notin J \text{ implies } \langle I, J\rangle \in \mathcal{S} \ . \qquad (C_{H^-})$$

Furthermore, the second condition in Lemma 8 implies that every $p \in B(r)^+$ satisfies the following condition:

$$p \notin I \text{ and } J \cap H(r)^+ \neq \emptyset \text{ implies } \langle I, J \rangle \in \mathcal{S} \ . \qquad (C_{B+})$$

These observations lead to the following results:

Lemma 10. *An atom p belongs to $B(r)^+$ if and only if for all $\langle I, J \rangle \in \mathcal{I}^{\mathsf{SE}}$, the conditions ($C_{H-}$) and ($C_{B+}$) are satisfied. An atom p belongs to $H(r)^-$ if and only if it does not belong to $B(r)^+$ and for all $\langle I, J \rangle \in \mathcal{I}^{\mathsf{SE}}$, the condition ($C_{H-}$) is satisfied.*

Together, the two Lemmas above are sufficient to reconstruct a canonical rule from its set of SE-models. The following definition sums up these results by introducing the notion of a rule induced by a set of SE-interpretations:

Definition 11 (Rule Induced by a Set of SE-Interpretations)
Let $\mathcal{S}$ be a set of SE-interpretations.

*An atom p is called an $\mathcal{S}$-*negative-body atom *if every SE-interpretation $\langle I, J \rangle$ with $p \in J$ belongs to $\mathcal{S}$. An atom p is called an $\mathcal{S}$-*positive-head atom *if it is not an $\mathcal{S}$-negative-body atom and every SE-interpretation $\langle I, J \rangle$ with $p \in I$ belongs to $\mathcal{S}$.*

*An atom p is called an $\mathcal{S}$-*positive-body atom *if every SE-interpretation $\langle I, J \rangle$ with $p \notin J$ belongs to $\mathcal{S}$, and every SE-interpretation $\langle I, J \rangle$ with $p \notin I$ and J containing some $\mathcal{S}$-positive-head atom also belongs to $\mathcal{S}$. An atom p is called an $\mathcal{S}$-*negative-head atom *if it is not an $\mathcal{S}$-positive-body atom and every SE-interpretation $\langle I, J \rangle$ with $p \notin J$ belongs to $\mathcal{S}$.*

The sets of all $\mathcal{S}$-negative-body, $\mathcal{S}$-positive-head, $\mathcal{S}$-positive-body and $\mathcal{S}$-negative-head atoms are denoted by $B(\mathcal{S})^-$, $H(\mathcal{S})^+$, $B(\mathcal{S})^+$ and $H(\mathcal{S})^-$, respectively. The rule induced by $\mathcal{S}$*, denoted by* $\mathsf{rule}(\mathcal{S})$*, is defined as follows: If $\mathcal{S} = \mathcal{I}^{\mathsf{SE}}$, then* $\mathsf{rule}(\mathcal{S})$ *is ε; otherwise,* $\mathsf{rule}(\mathcal{S})$ *is of the form*

$$H(\mathcal{S})^+; \sim H(\mathcal{S})^- \leftarrow B(\mathcal{S})^+, \sim B(\mathcal{S})^- .$$

The main property of induced rules is that every canonical rule is induced by its own set of SE-models and can thus be "reconstructed" from its set of SE-models. This follows directly from Definition 11 and Lemmas 9 and 10.

Theorem 12. *For every canonical rule r,* $\mathsf{rule}(\mathsf{mod}_{\mathsf{SE}}(r)) = r$.

This result, together with Theorem 7, has a number of consequences. First, for any rule r, the canonical rule $\mathsf{can}(r)$ is induced by the set of SE-models of r.

Corollary 13. *For every rule r,* $\mathsf{rule}(\mathsf{mod}_{\mathsf{SE}}(r)) = \mathsf{can}(r)$.

Furthermore, Theorem 12 directly implies that for two different canonical rules r_1, r_2 we have $\mathsf{rule}(\mathsf{mod}_{\mathsf{SE}}(r_1)) = r_1$ and $\mathsf{rule}(\mathsf{mod}_{\mathsf{SE}}(r_2)) = r_2$, so $\mathsf{mod}_{\mathsf{SE}}(r_1)$ and $\mathsf{mod}_{\mathsf{SE}}(r_2)$ must differ.

Corollary 14. *No two different canonical rules are SE-equivalent.*

Finally, the previous Corollary together with Theorem 7 imply that for every rule there not only exists an SE-equivalent canonical rule, but this rule is also unique.

Corollary 15. *Every rule is SE-equivalent to exactly one canonical rule.*

3.3 Sets of SE-Interpretations Representable by a Rule

Naturally, not all sets of SE-interpretations correspond to a single rule, otherwise any program could be reduced to a single rule. The conditions under which a set of SE-interpretations is rule-representable are worth examining.

A set of SE-models $\mathcal{S}$ of a program is always *well-defined*, i.e. whenever $\mathcal{S}$ contains $\langle I, J \rangle$, it also contains $\langle J, J \rangle$. Moreover, for every well-defined set of SE-interpretations $\mathcal{S}$ there exists a program $\mathcal{P}$ such that $\mathcal{S} = \mathsf{mod}_{\mathsf{SE}}(\mathcal{P})$ [10].

We offer two approaches to find a similar condition for the class of rule-representable sets of SE-interpretations. The first is based on induced rules defined in the previous Subsection, while the second is formulated using lattice theory and is a consequence of Lemma 8.

The first characterisation follows from two properties of the $\mathsf{rule}(\cdot)$ transformation. First, it can be applied to any set of SE-interpretations, even those that are not rule-representable. Second, if $\mathsf{rule}(\mathcal{S}) = r$, then it holds that $\mathsf{mod}_{\mathsf{SE}}(r)$ is a subset of $\mathcal{S}$.

Lemma 16. *The set of all SE-models of a canonical rule r is the least among all sets of SE-interpretations $\mathcal{S}$ such that* $\mathsf{rule}(\mathcal{S}) = r$*.*

Thus, to verify that $\mathcal{S}$ is rule-representable, it suffices to check that all interpretations from $\mathcal{S}$ are models of $\mathsf{rule}(\mathcal{S})$.

The second characterisation follows from Lemma 8 which tells us that if $\mathcal{S}$ is rule-representable, then its complement consists of SE-interpretations $\langle I, J \rangle$ following a certain pattern. Their second component J always contains a fixed set of atoms and is itself contained in another fixed set of atoms. Their first component I satisfies a similar property, but only if a certain further condition is satisfied by J. More formally, for the sets

$$I^{\perp} = B(r)^{+}, \quad I^{\top} = \mathcal{L} \setminus H(r)^{+}, \quad J^{\perp} = H(r)^{-} \cup B(r)^{+}, \quad J^{\top} = \mathcal{L} \setminus B(r)^{-},$$

it holds that all SE-interpretations from the complement of $\mathcal{S}$ are of the form $\langle I, J \rangle$ where $J^{\perp} \subseteq J \subseteq J^{\top}$ and either $J \subseteq I^{\top}$ or $I^{\perp} \subseteq I \subseteq I^{\top}$. It turns out that this also holds vice versa: if the complement of $\mathcal{S}$ satisfies the above property, then $\mathcal{S}$ is rule-representable. Furthermore, to accentuate the particular structure that arises, we can substitute the condition $J^{\perp} \subseteq J \subseteq J^{\top}$ with saying that J belongs to a convex sublattice of $\mathcal{I}$[1]. A similar substitution can be performed for I, yielding:

Theorem 17. *Let $\mathcal{S}$ be a set of SE-interpretations. Then the following conditions are equivalent:*

1. *The set of SE-interpretations $\mathcal{S}$ is rule-representable.*
2. *All SE-interpretations from $\mathcal{S}$ are SE-models of* $\mathsf{rule}(\mathcal{S})$*.*
3. *There exist convex sublattices L_1, L_2 of $\langle \mathcal{I}, \subseteq \rangle$ such that the complement of $\mathcal{S}$ relative to $\mathcal{I}^{\mathsf{SE}}$ is equal to*

$$\left\{ \langle I, J \rangle \in \mathcal{I}^{\mathsf{SE}} \;\middle|\; I \in L_1 \land J \in L_2 \right\} \cup \left\{ \langle I, J \rangle \in \mathcal{I}^{\mathsf{SE}} \;\middle|\; J \in L_1 \cap L_2 \right\} .$$

[1] A sublattice L of L' is *convex* if $c \in L$ whenever $a, b \in L$ and $a \leq c \leq b$ holds in L'. For more details see e.g. [22].

4 Discussion

The presented results mainly serve to facilitate the transition back and forth between a rule and the set of its SE-models. They also make it possible to identify when a given set of SE-models is representable by a single rule. We believe that in situations where information on literal dependencies, expressed in individual rules, is essential for defining operations on logic programs, the advantages of dealing with rules on the level of semantics instead of on the level of syntax are significant. The semantic view takes care of stripping away unnecessary details and since the introduced notions and operators are defined in terms of semantic objects, it should be much easier to introduce and prove their semantic properties.

These results can be used for example in the context of program updates to define an update semantics based on the *rule rejection principle* [4] and operating on *sets of sets of SE-models*. Such a semantics can serve as a bridge between syntax-based approaches to rule updates, and the principles and semantic distance measures known from the area of Belief Change. The next steps towards such a semantics involve a definition of the notion of support for a literal by a set of SE-models (of a rule). Such a notion can then foster a better understanding of desirable properties for semantic rule update operators.

On a different note, viewing a logic program as the *set of sets of SE-models* of rules inside it leads naturally to the introduction of the following new notion of program equivalence:

Definition 18 (Strong Rule Equivalence). *Programs $\mathcal{P}_1, \mathcal{P}_2$ are* SR-equivalent*, denoted by $\mathcal{P}_1 \equiv_{\mathsf{SR}} \mathcal{P}_2$, if*

$$\{ \, \mathsf{mod}_{\mathsf{SE}}(r) \mid r \in \mathcal{P}_1 \cup \{ \varepsilon \} \, \} = \{ \, \mathsf{mod}_{\mathsf{SE}}(r) \mid r \in \mathcal{P}_2 \cup \{ \varepsilon \} \, \} \ .$$

Thus, two programs are SR-equivalent if they contain the same rules, modulo the SE-model semantics. We add ε to each of the two programs in the definition so that presence or absence of tautological rules in a program does not influence program equivalence. SR-equivalence is stronger than strong equivalence, in the following sense:

Definition 19 (Strength of Program Equivalence). *Let $\equiv_1, \equiv_2$ be equivalence relations on the set of all programs. We say that $\equiv_1$* is at least as strong as *$\equiv_2$, denoted by $\equiv_1 \succeq \equiv_2$, if $\mathcal{P}_1 \equiv_1 \mathcal{P}_2$ implies $\mathcal{P}_1 \equiv_2 \mathcal{P}_2$ for all programs $\mathcal{P}_1, \mathcal{P}_2$. We say that $\equiv_1$* is stronger than *$\equiv_2$, denoted by $\equiv_1 \succ \equiv_2$, if $\equiv_1 \succeq \equiv_2$ but not $\equiv_2 \succeq \equiv_1$.*

Thus, using the notation of the above definition, we can write $\equiv_{\mathsf{SR}} \succ \equiv_{\mathsf{S}}$, where $\equiv_{\mathsf{S}}$ denotes the relation of strong equivalence. An example of programs that are strongly equivalent, but not SR-equivalent is $\mathcal{P} = \{ p., q. \}$ and $\mathcal{Q} = \{ p., q \leftarrow p. \}$, which in many cases need to be distinguished from one another. We believe that this notion of program equivalence is much more suitable for cases when the dependency information contained in a program is of importance.

In certain cases, however, SR-equivalence may be too strong. For instance, it may be desirable to treat programs such as $\mathcal{P}_1 = \{ p \leftarrow q. \}$ and $\mathcal{P}_2 = \{ p \leftarrow q., p \leftarrow q, r. \}$ in the same way because the extra rule in $\mathcal{P}_2$ is just a weakened version of the rule in $\mathcal{P}_1$.

For instance, the notion of *update equivalence* introduced in [23], which is based on a particular approach to logic program updates, considers programs $\mathcal{P}_1$ and $\mathcal{P}_2$ as equivalent because the extra rule in $\mathcal{P}_2$ cannot influence the result of any subsequent updates. Since these programs are not SR-equivalent, we also introduce the following notion of program equivalence, which in terms of strength falls between strong equivalence and SR-equivalence.

Definition 20 (Strong Minimal Rule Equivalence). *Programs $\mathcal{P}_1, \mathcal{P}_2$ are* SMR-*-equivalent, denoted by $\mathcal{P}_1 \equiv_{\mathsf{SMR}} \mathcal{P}_2$, if*

$$\min \{ \mathsf{mod}_{\mathsf{SE}}(r) \mid r \in \mathcal{P}_1 \cup \{ \varepsilon \} \} = \min \{ \mathsf{mod}_{\mathsf{SE}}(r) \mid r \in \mathcal{P}_2 \cup \{ \varepsilon \} \} \quad ,$$

where $\min \mathcal{S}$ denotes the set of subset-minimal elements of $\mathcal{S}$.

In order for programs to be SMR-equivalent, they need not contain exactly the same rules (modulo strong equivalence), it suffices if rules with subset-minimal sets of SE-models are the same (again, modulo strong equivalence). Certain programs, such as $\mathcal{P}_1$ and $\mathcal{P}_2$ above, are not SR-equivalent but they are still SMR-equivalent.

Related to this is the very strong notion of equivalence which was introduced in [2]:

Definition 21 (Strong Update Equivalence, c.f. Definition 4.1 in [2]). *Two programs $\mathcal{P}_1$, $\mathcal{P}_1$ are SU-equivalent, denoted by $\mathcal{P}_1 \equiv_{\mathsf{SU}} \mathcal{P}_2$, if for any programs $\mathcal{Q}$, $\mathcal{R}$ it holds that the program $((\mathcal{P}_1 \setminus \mathcal{Q}) \cup \mathcal{R})$ has the same answer sets as the program $((\mathcal{P}_2 \setminus \mathcal{Q}) \cup \mathcal{R})$.*

Two programs are strongly update equivalent only under very strict conditions – it is shown in [2] that two programs are SU-equivalent if and only if their symmetric difference contains only SE-tautological rules. This means that programs such as $\mathcal{Q}_1 = \{ {\sim}p. \}$, $\mathcal{Q}_2 = \{ \leftarrow p. \}$ and $\mathcal{Q}_3 = \{ {\sim}p \leftarrow p. \}$ are considered to be mutually non-equivalent, even though the rules they contain are mutually SE-equivalent. This may be seen as too sensitive to rule syntax.

The following result formally establishes the relations between the discussed notions of program equivalence:

Theorem 22. *SU-equivalence is stronger than SR-equivalence, which itself is stronger than SMR-equivalence, which in turn is stronger than strong equivalence. That is,*

$$\equiv_{\mathsf{SU}} \succ \equiv_{\mathsf{SR}} \succ \equiv_{\mathsf{SMR}} \succ \equiv_{\mathsf{S}} \quad .$$

The other notion of program equivalence introduced in [2], *strong update equivalence on common rules*, or SUC-equivalence, is incomparable in terms of strength to our new notions of equivalence. On the one hand, SR- and SMR-equivalent programs such as $\{ {\sim}p. \}$ and $\{ {\sim}p., \leftarrow p. \}$ are not SUC-equivalent. On the other hand, programs such as $\{ p., q \leftarrow p. \}$ and $\{ q., p \leftarrow q. \}$ are neither SR- nor SMR-equivalent, but they are SUC-equivalent. We believe that both of these examples are more appropriately treated by the new notions of equivalence.

The introduction of canonical rules, which form a set of representatives of rule equivalence classes induced by SE-models, also reveals the exact expressivity of SE-model semantics with respect to a single rule. From their definition we can see that

SE-models are capable of distinguishing between any pair of rules, except for (1) a pair of rules that only differ in the number of repetitions of literals in their heads and bodies; (2) an integrity constraint and a rule whose head only contains negative literals. We believe that in the former case, there is little reason to distinguish between such rules and so the transition from rules to their SE-models has the positive effect of stripping away of unnecessary details. However, the latter case has more serious consequences. Although rules such as

$$\sim p \leftarrow q. \qquad \text{and} \qquad \leftarrow p, q.$$

are usually considered to carry the same meaning, some existing work suggests that they should be treated differently – while the former rule gives a reason for atom p to become false whenever q is true, the latter rule simply states that the two atoms cannot be true at the same time, without specifying a way to resolve this situation if it were to arise [4,8]. If we view a rule through the set of its SE-models, we cannot distinguish these two kinds of rules anymore. Whenever this is important, either *strong update equivalence* is used, which is perhaps *too* sensitive to the syntax of rules, or a new characterisation of Answer-Set Programming needs to be discovered, namely one that is not based on the logic of Here-and-There [16,17].

Acknowledgement

We would like to thank Han The Anh, Matthias Knorr and the anonymous reviewers for their comments that helped to improve the paper. Martin Slota is supported by FCT scholarship SFRH / BD / 38214 / 2007.

References

1. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2(4), 526–541 (2001)
2. Inoue, K., Sakama, C.: Equivalence of logic programs under updates. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 174–186. Springer, Heidelberg (2004)
3. Damásio, C.V., Pereira, L.M., Schroeder, M.: REVISE: Logic programming and diagnosis. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 354–363. Springer, Heidelberg (1997)
4. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: Dynamic updates of non-monotonic knowledge bases. The Journal of Logic Programming 45(1-3), 43–70 (2000)
5. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: On properties of update sequences based on causal rejection. Theory and Practice of Logic Programming 2(6), 721–777 (2002)
6. Sakama, C., Inoue, K.: An abductive framework for computing knowledge base updates. Theory and Practice of Logic Programming 3(6), 671–713 (2003)
7. Zhang, Y.: Logic program-based updates. ACM Transactions on Computational Logic 7(3), 421–472 (2006)
8. Alferes, J.J., Banti, F., Brogi, A., Leite, J.A.: The refined extension principle for semantics of dynamic logic programming. Studia Logica 79(1), 7–32 (2005)

9. Delgrande, J.P., Schaub, T., Tompits, H.: A preference-based framework for updating logic programs. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 71–83. Springer, Heidelberg (2007)
10. Delgrande, J.P., Schaub, T., Tompits, H., Woltran, S.: Belief revision of logic programs under answer set semantics. In: Brewka, G., Lang, J. (eds.) Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning, Sydney, Australia, September 16-19, pp. 411–421. AAAI Press, Menlo Park (2008)
11. Delgrande, J.P.: A Program-Level Approach to Revising Logic Programs under the Answer Set Semantics. In: Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming Special Issue, vol. 10(4-6), pp. 565–580 (2010)
12. Gärdenfors, P.: Belief Revision: An Introduction. In: Belief Revision, pp. 1–28. Cambridge University Press, Cambridge (1992)
13. Slota, M., Leite, J.: On semantic update operators for answer-set programs. In: Coelho, H., Studer, R., Wooldridge, M. (eds.) Proceedings of the 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 957–962. IOS Press, Amsterdam (2010)
14. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann, San Francisco (1988)
15. Dix, J.: A classification theory of semantics of normal logic programs: II. Weak properties. Fundamenta Informaticae 22(3), 257–288 (1995)
16. Łukasiewicz, J.: Die Logik und das Grundlagenproblem. In: Les Entretiens de Zürich sue les Fondements et la méthode des sciences mathématiques 1938, Zürich, pp. 82–100 (1941)
17. Pearce, D.: A new logical characterisation of stable models and answer sets. In: Dix, J., Przymusinski, T.C., Moniz Pereira, L. (eds.) NMELP 1996. LNCS, vol. 1216, pp. 57–70. Springer, Heidelberg (1997)
18. Turner, H.: Strong equivalence made easy: nested expressions and weight constraints. Theory and Practice of Logic Programming 3(4-5), 609–622 (2003)
19. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) Proceedings of the 5th International Conference and Symposium on Logic Programming, August 15-19, pp. 1070–1080. MIT Press, Washington (1988)
20. Inoue, K., Sakama, C.: Negation as failure in the head. Journal of Logic Programming 35(1), 39–78 (1998)
21. Cabalar, P., Pearce, D., Valverde, A.: Minimal logic programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 104–118. Springer, Heidelberg (2007)
22. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)
23. Alexandre Leite, J.: Evolving Knowledge Bases. Frontiers of Artificial Intelligence and Applications, vol. 81, xviii + 307 p. IOS Press, Amsterdam (2003); Hardcover

What Are the Necessity Rules in Defeasible Reasoning?

Ho-Pun Lam[1,2] and Guido Governatori[2]

[1] School of Information Technology and Electrical Engineering
The University of Queensland, Brisbane, Australia
[2] NICTA*, Queensland Research Laboratory, Brisbane, Australia

Abstract. This paper investigates a new approach for computing the inference of defeasible logic. The algorithm proposed can substantially reduced the theory size increase due to transformations while preserving the representation properties in different variants of DL. Experiments also show that our algorithm outperform traditional approach by several order of amplitudes.

1 Introduction

Defeasible reasoning [1] is a simple rule-based skeptical approach. This approach offers two main advantages over other mainstream nonmonotonic reasoning formalisms: (i) low computation complexity (linear time w.r.t. the size of a theory) [2] and (ii) its build-in preference handling facilities allowing one to derive plausible conclusions from incomplete and contradictory information in a natural and declarative way.

The strategy to compute the extension of a defeasible is to apply a series of preprocessing transformations that transform a defeasible theory into an equivalent theory without superiority relations and defeaters and then applies the reasoning algorithm [2] to the transformed theory. For instance, consider the example below:

Example 1. Let us consider the defeasible theory:

$$\begin{array}{lll} r_1 : \Rightarrow a & r'_1 : a \Rightarrow c & r_1 > r_2 \\ r_2 : \Rightarrow \neg a & r'_2 : \Rightarrow \neg c & \end{array}$$

The transformation of the above theory is:

$$\begin{array}{lll} r_1.a : \Rightarrow \neg inf^+(r_1) & r'_1.a : a \Rightarrow \neg inf^+(r'_1) & \\ r_1.c : \neg inf^+(r_1) \Rightarrow a & r'_1.c : \neg inf^+(r'_1) \Rightarrow c & \text{and} \quad s_1^+ : \neg inf^+(r_2) \Rightarrow inf^+(r_1) \\ r_2.a : \Rightarrow \neg inf^+(r_2) & r'_2.a : \Rightarrow \neg inf^+(r'_2) & \phantom{\text{and}} \quad s_1^- : \neg inf^+(r_2) \Rightarrow inf^-(r_1) \\ r_2.c : \neg inf^+(r_2) \Rightarrow \neg a & r'_2.c : \neg inf^+(r'_2) \Rightarrow \neg c & \end{array}$$

It is clear that these transformations were designed to provide incremental transformations to the theory, and systematically introduces new literals and rules to emulate the features removed [3]. However, as pointed out in [4], such transformations are profligate in their introduction of propositions and generation of rules, which would result in an increase in theory size by at most a factor of 12.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

In addition, such transformations cannot preserve the representation properties of DL in different variants[1]. For instance, consider the theory as shown in Example [1]. Under ambiguity propagation variant the conclusions derived should be $+\partial\neg a$, $-\partial a$, $+\partial\neg c$, $-\partial c$. However, in the transformed theory, as the superiority relation is removed, the support of a in $r_1.c$ cannot be blocked, which subsequently propagated and supported the conclusions of $r'_1.a$ and $r'_1.c$. Hence the conclusions derived in the transformed theory becomes $+\partial\neg a, -\partial a, -\partial\neg c, -\partial c$, instead of the one desired.

Due to the deficiencies above and the inability of current reasoning algorithm to handle superiority relation directly, the focus of this paper is on finding the necessity rules in deriving the conclusions associated with superiority relations. Thus the aim is on finding conditions under which rules redundant under superiority relations and can be removed from the theory. We believe that our approach, in general, can also be applied to other rule-based nonmonotonic formalisms containing preference operator(s) that describes the relative strength of rules, such as preference logic.

2 Basics of Defeasible Logic

In this section we provide a short outline of DL and the construction of variants capturing different intuitions of non-monotonic reasoning based on modular and parametrized definition of the proof theory of the logic. For the full details, please refer to [3,6,7].

A defeasible theory D is a triple $(F,R,>)$ where F and R are finite set of facts and rules respectively, and $>$ is an acyclic superiority relation on R. Facts are logical statements describing indisputable facts, represented by (atomic) propositions (i.e. literals). A rule r describes the relations between a set of literals (the antecedent $A(r)$, which can be empty) and a literal (the consequence $C(r)$). DL supports three kinds of rules: *strict rules* ($r : A(r) \rightarrow C(r)$), *defeasible rules* ($r : A(r) \Rightarrow C(r)$) and *defeaters* ($r : A(r) \leadsto C(r)$). *Strict rules* are rules in classical sense, the conclusion follows every time the antecedents hold; a *defeasible rule* is allowed to assert its conclusion in case there is no contrary evidence to it. *Defeaters* cannot support conclusions but can provide contrary evidence to them. The *superiority relation* describes the relative strength of rules, and is used to obtain a conclusion where there are applicable conflicting rules.

DL is able to distinguish positive conclusions from negative conclusions, that is literals that can be proved and literals that are refuted. In addition, it is able to determine the strength of a conclusion, i.e., whether something is concluded using only strict rules and facts or whether we have a defeasible conclusion, a conclusion can be retracted if more evidence is provided. According, for a literal p we have the following four types of conclusions, called tagged literals: $+\Delta p$ (p is definitely provable), $-\Delta p$ (p is definitely refuted), $+\partial p$ (p is defeasible provable), and $-\partial p$ (p is defeasible refuted).

At the heart of DL we have its proof theory that tells us how to derive tagged literals. A *proof* is a sequence of tagged literals obeying proof conditions corresponding to inference rules. The inference rules establish when we can add a literal at the end of a sequence of tagged literals based on conditions on the elements of a theory and the

[1] Several variants (such as *ambiguity propagation (AP)*, *well-founded semantics (WF)*) of DL have been proposed to capture the intuitions of different non-monotonic reasoning formalism. Readers interested please refer to [5] for details.

previous tagged literals in the sequence. The structure of the proof conditions has an argumentation flavour. For example, to prove $+\partial p$:

Phase 1: There is an applicable rule for p and
Phase 2: For every rule for $\neg p$ (the complement of p) either
  Sub-Phase 1: the rule is discarded, or
  Sub-Phase 2: the rule is defeated by a (stronger) rule for p

The notion of a rule being applicable means that all the antecedents of the rule are provable (with the appropriate strength); a rule is discarded if at least one of the antecedents is refuted (with the appropriate strength), and finally a rule is defeated, if there is a (stronger) rule for the complement of the conclusion that is applicable (again with the appropriate strength).

The above structure enables us to define several variants of DL [5] –such as ambiguity blocking and ambiguity propagation– by giving different parameters (i.e., this is what we mean 'with the appropriate strength' in the previous paragraph).

3 Inferiorly Defeated Rules

As mentioned before, the *superiority relation* in DL is used to define the preference, or relative strength, of rules, i.e., it provides information about which rules can overrule which other rules. And in some variants, such as ambiguity propagation, the superiority relation also provides information on whether the consequences of rules are supported or not. Based on these, we introduce the notion of *superiority chain*.

Definition 1. *A* superiority chain *is a superiority relation hierarchy such that, apart form the first and last element of the chain, there exists a superiority relation between rules r_k and r_{k+1}:*

$$r_1 > r_2 > \cdots > r_n$$

where n is the length of the chain, and $C(r_k) = \neg C(r_{k+1})$, $\forall 1 \leq k < n$.

Notice that the superiority relation is not transitive, i.e., unless otherwise specified, there exists no superiority relation between a rule r and another rule in the chain. Consider the theory in example 2 below: r_1 and r_4 are in the same superiority chain but r_1 is not superior to r_4, the consequence of r_1 cannot be used to overrule the consequence of r_4.

Lemma 1. *Let $D = (\emptyset, R, >)$[2] be a defeasible theory (in regular form) over a language $\mathscr{L}_D$. Then a rule $r \in R_{sd}[q]$ is* inferiorly defeated *if $\exists s \in R_{sd}[\neg q] : A(s) = \emptyset$ and $s > r$.*

The key observation of this lemma is that *irrespective to whethter an inferiorly defeated rule r is derivable or not, its conclusion is going to be overruled by a superior rule s.* Consider again the theory in Example 1. Since r_1 and r_2 are both derivable and r_2 is superior than r_1, r_1 is inferiorly defeated and its conclusion is overridden by r_2. So, under this situation, r_1 is redundant and cannot be used to derive any positive conclusion. Removing it from the theory and falsifying its conclusion does not affect the conclusions derived. And the same applies even when $A(r_1) \neq \emptyset$.

[2] A defeasible theory $D = (F, R, >)$ can be transformed to an equivalent theory $D' = (\emptyset, R', >')$ (without fact) using the algorithm proposed in [3].

However, the example above is just oversimplified. Consider the example below:

Example 2. Let us consider the defeasible theory (D):

$$r_1 :\Rightarrow a \quad r_2 :\Rightarrow \neg a \quad r_3 :\Rightarrow a \quad r_4 :\Rightarrow \neg a \qquad \text{and } r_1 > r_2 > r_3 > r_4$$

All rules above are derivable but r_2, r_3 and r_4 are inferiorly defeated and r_1 is the only rule that can be used to derive positive conclusion. So, the conclusions inferred should be $+\partial a, -\partial\neg a$. However, if we remove an inferiorly defeated rule arbitrarily, say r_3, then the theory will becomes (D'):

$$r_1 :\Rightarrow a \qquad r_2 :\Rightarrow \neg a \qquad r_4 :\Rightarrow \neg a \qquad r_1 > r_2$$

Then, only r_2 is inferiorly defeated and the conclusions derived will become $-\partial a, -\partial\neg a$, implying that $D \not\equiv D'$. Hence a rule is inferiorly defeated is not an adequate condition all by itself to enable it to be removed from the theory without changing the conclusions. If we take into account a line of superiority chain then additional conditions are needed.

4 Statement of Results

To recap, our main focus lies in characterizing a set inferiorly defeated rules that cannot be used to derive positive conclusions and can be removed from the theory without changing its conclusions.

Definition 2. *Let $D = (\emptyset, R, >)$ be a defeasible theory (in regular form) over a language $\mathscr{L}_D$. Then, R_{infd} is the set of inferiorly defeated rules in D with number of weaker rules equal to zero. That is, $\forall r \in R_{infd}[q]$, $\nexists s \in R[\neg q]$ s.t. $r > s$ (i.e., r is the* weakest rules *along the line of superiority chain).*

Theorem 1. *Let $D = (\emptyset, R, >)$ be a defeasible theory (in regular form), and $r \in R_{infd}$. Let $D' = (\emptyset, R \setminus \{r\}, >')$ be the reduct of D, denoted by $reduct(D)$, where $>'$ is defined by the following condition:*

$$\forall s \in R,\ A(s) = \emptyset,\ s > r\ (>' \Leftrightarrow > \setminus \{s > r\})$$

Then $D \equiv D'$. In addition, $-\partial q$ can be derived if $R[q] = \emptyset$ after the removal of r.

This theorem looks simple but plays a fundamental role in differentiating the set of necessity rules that are required in the inference process from the set of rules that are redundant and can be removed from the theory. From the theorem, we can conclude that if an inferiorly defeated rule is the weakest rule of a superiority chain, then removing it from the theory will not cause any undesired effects to the inference process. For instance, consider the theory again in Example 2 and apply the theorem recursively, then, r_4, r_3 and r_2 will be removed subsequently leaving r_1 as the only rule in the superiority chain, which thus give us the results: $+\partial a$ and $\partial\neg a$, as desired.

Therefore, by incorporating the theorem into the inference process, the transformation of eliminating the superiority relation is no longer necessary, which thus reduced the transformation time as well as the number of propositions introduced and rules generated. In addition, upon satisfying the conditions stated above, an inferiorly defeated rule can be removed from the theory immediately even without knowing any information about the provability of its body, which can further enhance the performance of the reasoning engine. Furthermore, the result is general and it applies to the ambiguity blocking (AB), ambiguity propagating (AP) and well-founded (WF) variants of DL.

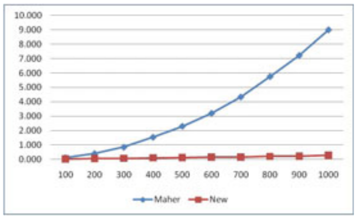
4.1 Implementation and Some Experimental Results

The above theorem has been incorporated into the inference algorithm proposed in [2] (for AB) and [8] (for AP and WF) and is implemented in the latest version of SPINdle [9]. The implementation is tested using the UAV navigation theory described in [10] (consists of about a hundred rules and 40 superiority relations) and the superioirty relation test theories generated by a tool that comes with Deimos [11]. The test has been measured on a Pentium 4 PC (3GHz) with Window XP and 2GB main memory. Table 1 and Figure 1 below show the performance result and their memory usage.

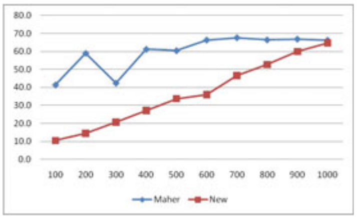
As shown, our approach outperforms the traditional approach under the same model complexity. Even with theory 1000 rules and 500 superiority relation, our approach can complete the whole inference process in less than half a second (289ms), while it takes 9 for the approach based on a transformation to remove the superiority relation. In terms of memory usage, our approach only increases linearly with the original theory size while the traditional approach a significantly larger memory size.

Table 1. Reasoning time used in UAV navigation theory

Algorithm	Superiority relation removal (ms)	Conclusions generation (ms)	Total Reasoning Time (ms)	Memory Usage (MB)
Traditional	47	156	203	1.52
New	-	93	93	1.36



(a) Reasoning time (sec)



(b) Memory usage (MB)

Fig. 1. Superiority relations test theories - Reasoning time and Memory usage (with different theories size)

5 Related Works and Conclusions

A number of defeasible logic reasoners and systems have been proposed in recent years to cover different variants of defeasible reasoning and other intuitions of nonmonotonic reasoning. Most of the implementations rely on either transformations, or translation to other logic formalism (such as extended logic programming) [12,13] and carry out the inference process using the underlying reasoning engine. However, as mentioned before, the representation properties of DL cannot be preserved after transformation. Also, most of the system are query based and do not compute diectly the extension of a theory, which may leads to some counterintuitive result [14]. The meta-program approach is at the foundation of the approaches based on transformation in other formalism. DR-Prolog [12], which directly implement the meta-programs of [5], provides a Semantic Web enabled implementation of DL.

In this paper, we have presented a theorem that allow us to reason on a defeasible theory without removing the superiority relation. The essence of this method lies in the ability in identifying inferiorly defeated rules that are redundant and can be removed from the theory, which help in preserving the representation properties of defeasible logic across different variants and simplified the work in subsequent processes. Our result shows that the performance gains are significant. As a future work, we will work on studying the relation of conclusions between different variants, with special interest in ambiguity propagation and well-founded semantics.

References

1. Nute, D.: Defeasible logic. In: Gabbay, D., Hogger, C. (eds.) Handbook of Logic for Artificial Intelligence and Logic Programming, vol. III, pp. 353–395. Oxford University Press, Oxford (1994)
2. Maher, M.J.: Propositional defeasible logic has linear complexity. Theory and Practice of Logic Programming 1(6), 691–711 (2001)
3. Antoniou, G., Billington, D., Governatori, G., Maher, M.J.: Representation results for defeasible logic. ACM Transactions on Computational Logic 2(2), 255–286 (2001)
4. Bryant, D., Krause, P.: A review of current defeasible reasoning implementations. Knowl. Eng. Rev. 23(3), 227–260 (2008)
5. Antoniou, G., Billington, D., Governatori, G., Maher, M.J., Rock, A.: A family of defeasible reasoning logics and its implementation. In: Proc. ECAI 2000, pp. 459–463 (2000)
6. Antoniou, G., Billington, D., Governatori, G., Maher, M.J.: A flexible framework for defeasible logics. In: AAAI 2000, pp. 401–405. AAAI/MIT Press (2000)
7. Billington, D., Antoniou, G., Governatori, G., Maher, M.J.: An inclusion theorem for defeasible logic. ACM Transactions in Computational Logic 12(1) (2010)
8. Lam, H.-P., Governatori, G.: On the problem of computing ambiguity propagation and well-founded semantics in defeasible logic. In: Dean, M., Hall, J., Rotolo, A., Tabet, S. (eds.) RuleML 2010. LNCS, vol. 6403, pp. 119–127. Springer, Heidelberg (2010)
9. Lam, H.-P., Governatori, G.: The making of SPINdle. In: Governatori, G., Hall, J., Paschke, A. (eds.) RuleML 2009. LNCS, vol. 5858, pp. 315–322. Springer, Heidelberg (2009)
10. Lam, H.P., Thakur, S., Governatori, G., Sattar, A.: A model to coordinate uavs in urban environment using defeasible logic. In: Hu, Y.J., Yeh, C.L., Laun, W., Governatori, G., Hall, J., Paschke, A. (eds.) Proc. RuleML 2009 Challenge. CEUR Workshop Proceedings, vol. 549 (2009)
11. Maher, M.J., Rock, A., Antoniou, G., Billington, D., Miller, T.: Efficient defeasible reasoning systems. International Journal on Artificial Intelligence Tools 10(4), 483–501 (2001)
12. Antoniou, G., Bikakis, A.: DR-Prolog: A system for defeasible reasoning with rules and ontologies on the semantic web. IEEE Trans. Knowl. Data Eng. 19(2), 233–245 (2007)
13. Madalińska-Bugaj, E., Lukaszewicz, W.: Formalizing defeasible logic in cake. Fundam. Inf. 57(2-4), 193–213 (2003)
14. Antoniou, G.: A discussion of some intuitions of defeasible reasoning. In: Vouros, G.A., Panayiotopoulos, T. (eds.) SETN 2004. LNCS (LNAI), vol. 3025, pp. 311–320. Springer, Heidelberg (2004)

Partial Preferences and Ambiguity Resolution in Contextual Defeasible Logic*

Antonis Bikakis[1] and Grigoris Antoniou[2]

[1] University of Luxembourg
[2] Institute of Computer Science, FO.R.T.H., Greece

Abstract. Domains, such as Ambient Intelligence and Social Networks, are characterized by some common features including distribution of the available knowledge, entities with different backgrounds, viewpoints and operational environments, and imperfect knowledge. Multi-Context Systems (MCS) has been proposed as a natural representation model for such environments, while recent studies have proposed adding non-monotonic features to MCS to address the issues of incomplete, uncertain and ambiguous information. In previous works, we introduced a non-monotonic extension to MCS and an argument-based reasoning model that handle imperfect context information based on defeasible argumentation. Here we propose alternative variants that integrate features such as partial preferences, ambiguity propagating and team defeat, and study the relations between the different variants in terms of conclusions being drawn in each case.

1 Introduction

Multi-Context Systems (*MCS*, [1,2]) are logical formalizations of distributed context theories connected through a set of mapping rules, which enable information flow between different contexts. A *context* can be thought of as a logical theory - a set of axioms and inference rules - that models local knowledge. In order to address various imperfections in context, e.g. incompleteness, uncertainty, ambiguity, recent studies have proposed adding non-monotonic features to MCS. Two representative examples are: (*a*) the non-monotonic rule-based MCS framework [3], which supports default negation in the mapping rules; and (*b*) the multi-context variant of Default Logic, *ConDL* [4], which models bridge relations as default rules, in order to handle cases of mutually inconsistent contexts. Both approaches, though, do not provide ways to model the quality of imported knowledge, nor preference between different information sources, leaving potential conflicts that may arise during the interaction of contexts unresolved.

In previous works we proposed *Contextual Defeasible Logic* (*CDL*), a nonmonotonic extension to MCS, which integrates preferences and preference-based reasoning. CDL uses both strict and defeasible rules to model local knowledge, defeasible mapping rules, and a *per context* preference order on the set of contexts to resolve conflicts caused by the interaction of mutually inconsistent contexts. The representation model of CDL, an argumentation semantics and associated algorithms for distributed query

* This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.

evaluation were presented in [5], while its formal properties and relation with Defeasible Logic were analyzed in [6]. Finally, an extension of CDL with partial preference orders, was proposed in [7]. The main consequence of partial ordering is that conflicts among competing arguments cannot always be resolved. To address such cases, here we propose four different variants of CDL, each of which implements a different semantics for ambiguity resolution. The first two variants implement an *ambiguity blocking* behavior, according to which arguments that are rejected due to unresolved conflicts cannot support attacks against other arguments. The *ambiguity propagating* behavior, adopted by the other two variants, enables mutually attacking arguments to support attacks and block other arguments. The latter behavior results in fewer arguments being justified, or in other words, fewer conclusions being drawn. Both semantics have also been defined and implemented for Defeasible Logics [8] and defeasible argumentation [9]. Two of the variants also support *team defeat*. This feature is common in many prioritized nonmonotonic logics. Here, it enables arguments to form teams in order to defend themselves against attacks and support their common conclusions. In this way, it resolves some of the conflicts caused by the lack of sufficient preference information.

The rest of the paper is structured as follows. Section 2 describes the MCS-based representation model. Section 3 presents the four different variants of CDL and studies their properties. Finally, Section 4 provides a summary of the results and proposes some possible directions for future research.

2 Representation Model

In CDL, a defeasible MCS C is defined as a collection of context theories C_i. A context C_i is defined as a tuple (V_i, R_i, T_i), where V_i is the vocabulary used by C_i (a set of positive and negative literals), R_i is a set of rules, and T_i is a preference ordering on C.

R_i consists of a set of *local* rules and a set of *mapping* rules. The body of a local rule is a conjunction of *local* literals (literals that are contained in V_i), while its head contains a local literal. Local rules are classified into: (a) strict rules, $r_i^l : a_i^1, a_i^2, ...a_i^{n-1} \rightarrow a_i^n$, which express sound local knowledge and are interpreted in the classical sense: whenever the literals in the body of the rule are strict consequences of the local theory, then so is the conclusion of the rule; and (b) defeasible rules, $r_i^d : b_i^1, b_i^2, ...b_i^{n-1} \Rightarrow b_i^n$, used to express local uncertainty, in the sense that a defeasible rule cannot be applied to support its conclusion if there is adequate contrary evidence. Mapping rules are used to associate concepts used by different contexts. The body of each such rule is a conjunction of local and foreign literals, while its head contains a single local literal. To deal with ambiguities caused by the interaction of mutually inconsistent contexts, mapping rules are also modeled as defeasible rules: $r_i^m : a_i^1, a_j^2, ...a_k^{n-1} \Rightarrow a_i^n$.

Finally, each context C_i defines a partial preference ordering T_i on C to express its confidence in the knowledge it imports from other contexts. T_i is modeled as a directed acyclic graph, in which vertices represent system contexts and arcs represent preference relations between the contexts that label the connected vertices. A context C_j is preferred by C_i to context C_k, denoted as $C_j >^i C_k$, if there is a path from vertex labeled by C_k to vertex labeled by C_j in T_i.

3 Argumentation Semantics

In this section, we describe four different variants of CDL using argumentation semantics. The first two variants, described in sections 3.1-3.2, implement the *ambiguity blocking* behavior, while those described in 3.3-3.4 implement the *ambiguity propagating* one. The variants described in 3.2 and 3.4 implement the notion of *team defeat*.

3.1 Ambiguity Blocking Semantics

The CDL variant proposed in [7] is actually the one that implements the ambiguity blocking behavior. Here, we present again its main definitions.

An argument A for a literal p_i in context C_i is defined as a tuple (C_i, PT_{p_i}, p_i), where PT_{p_i} is the proof tree for p_i constructed using rules from R_i. There are two types of arguments: (a) *local arguments*, which use local rules only; and (b) *mapping arguments*, which use at least one mapping rule. Local arguments are classified into strict local arguments, which use strict rules only, and defeasible local arguments, which contain at least one defeasible rule. We denote as $Args_{C_i}$ the set of all arguments of C_i, and as $Args_C$ the set of all arguments in $C = \{C_i\}$: $Args_C = \bigcup_i Args_{C_i}$.

An argument A *attacks* a defeasible local or mapping argument B at p_i, if p_i is a conclusion of B, $\sim p_i$ is a conclusion of A, and for the subarguments of A, A' with conclusion $\sim p_i$, and of B, B' with conclusion p_i, it holds that: (a) B' is not a strict local argument; and either (b_1) A' is a local argument of C_i or (b_2) B' is a mapping argument of C_i and $\exists b_l \in B$, s. t. $\forall a_k \in A$ there is no path from C_k to C_l in T_i ($C_l \not>^i C_k$).

An *argumentation line* A_L for a literal p_i is a sequence of arguments in $Args_C$, constructed in steps as follows: In the first step one argument for p_i is added in A_L. In each next step, for each distinct literal q_j labeling a leaf node of the the arguments added in the previous step, one argument B with conclusion q_j is added, provided that there is no argument $D \neq B$ for q_j already in A_L. The argument added in the first step is called the *head argument* of A_L, and p_i is called the conclusion of A_L. If the number of steps required to build A_L is finite, then A_L is a finite argumentation line.

An argument A is *supported* by a set of arguments S if: (a) every proper subargument of A is in S; and (b) there is a finite argumentation line A_L with head A, such that every argument in $A_L - \{A\}$ is in S. An argument A is *undercut* by a set of arguments S if for every argumentation line A_L with head A, there is an argument B, such that B is supported by S, and B defeats a proper subargument of A or an argument in $A_L - \{A\}$.

An argument A is *acceptable* w.r.t a set of arguments S if: (a) A is a strict local argument; or (b) A is supported by S and every argument attacking A is undercut by S. Based on the concept of acceptable arguments, we define *justified arguments* and *justified literals*. Let C be a MCS. J_i^C is defined as follows: for $i = 0$, $J_i^C = \emptyset$; for $i > 0$: $J_i^C = \{A \in Args_C \mid A \text{ is acceptable w.r.t. } J_{i-1}^C\}$.

The set of *justified arguments* in a MCS C is $JArgs^C = \bigcup_{i=1}^{\infty} J_i^C$. A literal p_i is *justified* in C if it is the conclusion of an argument in $JArgs^C$. That a literal p_i is justified, it actually means that it is a logical consequence of C.

An argument A is *rejected* by a set of arguments T when A is undercut by T; or A is attacked by an argument supported by T. The set of *rejected arguments* in C (denoted as $RArgs^C$) is the set of arguments that are rejected by $JArgs^C$. A literal p_i is *rejected* in C if there is no argument in $Args^C - RArgs^C$ with conclusion p_i. That p_i is rejected means that we are able to prove that it cannot be drawn as a logical conclusion.

3.2 Ambiguity Blocking with Team Defeat

Implementing team defeat requires introducing some new notions in the framework. The notion of defeat describes the case that an argument is clearly preferred to its counter-arguments. Formally, an argument A *defeats* an argument B at q if A attacks B at q, and A is not attacked by B at $\neg q$. The notions of attack and defeat can also be used for sets of arguments. A group of argument S attacks (defeats) an argument B at q if there is at least one argument A in S, such that A attacks (defeats) B at $\neg q$. A set of arguments T *teams defeats* a set of arguments S at q if for every argument $B \in S$ with conclusion q, there is an argument $A \in T$, such that A defeats B at q. The definition of undercut should also be revised to take into account team defeat. A set of arguments S is *undercut* by a set of arguments T if there is an argument $A \in S$ such that for every argumentation line A_L with head A there is an argument B such that: (a) B is supported by T; (b) B attacks a proper subargument of A or an argument in $A_L - \{A\}$ at q; and (c) S does not team defeat B at $\sim q$. Finally, we introduce the notion of *supported set*. The *supported set* of a set of arguments S, denoted as $Supported(S)$, is the maximal set of arguments that are supported by S. Under ambiguity blocking semantics with team defeat, an argument A is *acceptable* w.r.t a set of arguments S if: (a) A is a strict local argument; or (b) A is supported by S and every set of arguments attacking A (at q) is either undercut by S or team defeated by $Supported(S)$ (at $\sim q$). A set of arguments S is *rejected* by a set of arguments T when: (a) S is undercut by T; or (b) there is an argument $A \in S$, such that A is attacked by $Supported(T)$ at one of its conclusions q, and S does not team defeat $Supported(T)$ at $\sim q$. An argument A is rejected by T if for every set of arguments S, such that $A \in S$, S is rejected by T.

3.3 Ambiguity Propagating without Team Defeat

To implement ambiguity propagating semantics we use the notion of *proper undercut* and new acceptability and refutability semantics. An argument A is *properly undercut* by a set of arguments S if for every argumentation line A_L with head A: there is an argument $B \in S$, such that B attacks a proper subargument of A or an argument in $A_L - \{A\}$. An argument A is *acceptable* w.r.t a set of arguments S if: (a) A is a strict local argument; or (b) A is supported by S and every argument attacking A is properly undercut by S. An argument A is *rejected* by a set of arguments T when: (a) A is attacked by an argument B, such that B is not properly undercut by T; or (b) for every argumentation line A_L with head A, there exists an argument B, such that B attacks an argument in $A_L - \{A\}$, and B is not properly undercut by T.

3.4 Ambiguity Propagating with Team Defeat

This last variant of CDL combines the ambiguity propagating semantics with the notion of team defeat. Under this semantics, an argument A is *acceptable* w.r.t a set of arguments S if: (a) A is a strict local argument; or (b) A is supported by S and every argument attacking A (at q) is either properly undercut by S or team defeated (at $\sim q$) by $Supported(S)$. An argument A is *rejected* by a set of arguments T when: (a) A is attacked by an argument B that is neither properly undercut by T nor team defeated by $Supported(T)$; or (b) for every argumentation line A_L with head A, there exists an argument B, such that B attacks an argument in $A_L - \{A\}$, and B is neither properly undercut by T nor team defeated by $Supported(T)$.

3.5 Properties

Theorems 1 to 3 hold for all variants of CDL. Theorem 1 refers to the monotonicity in J_i^C, while Theorems 2 and 3 refer to system consistency[1].

Theorem 1. *For any defeasible MCS C, the sequence J_i^C is monotonically increasing.*

Theorem 2. *For any defeasible MCS C, it holds that: (a) No argument in $Args_C$ is both justified and rejected; (b) No literal in C is both justified and rejected.*

Theorem 3. *For any defeasible MCS C, if $JArgs^C$ contains two arguments with complementary conclusions, then both are strict local arguments in $Args_C$.*

Theorem 4 describes the relations that hold between the four different variants with respect to the results that they produce. $J_{var}(C)$ and $R_{var}(C)$ denote, respectively, the sets of justified and rejected literals in a MCS C under the semantics of the variant described by index *var*, which may take one of the following values: (a) *ab,ntd* for the ambiguity blocking variant without team defeat; (b) *ab,td* for the ambiguity blocking variant with team defeat; (c) *ap,ntd* for the ambiguity propagating variant without team defeat; (d) *ap,td* for the ambiguity propagating variant with team defeat.

Theorem 4. *For any defeasible MCS C, the following relations hold:*

1. $J_{ap,ntd}(C) \subseteq J_{ap,td}(C) \subseteq J_{ab,td}(C)$
2. $J_{ap,ntd}(C) \subseteq J_{ab,ntd}(C)$
3. $R_{ap,ntd}(C) \supseteq R_{ap,td}(C) \supseteq R_{ab,td}(C)$
4. $R_{ap,ntd}(C) \supseteq R_{ab,ntd}(C)$

Theorem 5 describes a relation between partial preference ordering and total preference ordering MCS. The intuition behind this is that under ambiguity propagating semantics, the more preference information is available, the more conflicts between counter-arguments can be resolved, and the more arguments and literals can be justified (and less rejected). Assuming a MCS $C = \{C_i\}$, where each context $C_i = \{V_i, R_i, T_i^p\}$ uses a partial preference ordering T_i^p, we call $C^T = \{C_i^T\}$, $C_i^T = \{V_i, R_i, T_i^t\}$ a *preference extension* of C, if for every T_i^t it holds that T_i^t is a linear extension of T_i^p.

Theorem 5. *For any Defeasible MCS C and any preference extension of C, C^T, under the ambiguity propagating semantics it holds that :*

1. $J_{ap}(C) \subseteq J_{ap}(C^T)$
2. $R_{ap}(C) \supseteq R_{ap}(C^T)$

[1] The proofs for all theorems that appear in this section are available at `http://www.csd.uoc.gr/~bikakis/partialCDL.pdf`

4 Discussion

Contextual Defeasible Logic is a nonmonotonic extension to MCS integrating preference information to resolve potential ambiguities caused by the interaction of contexts. In this paper, we propose four different variants of CDL, which adopt different semantics for handling ambiguity, implementing different levels of skepticism. Ambiguity propagating is more skeptical than ambiguity blocking, while removing team defeat results in even more skeptical behavior. On the other hand, under ambiguity propagating semantics, the system is monotonic with respect to the available preference information in terms of number of conclusion being drawn. Therefore, choosing the proper variant depends to a large extent on the cost of an incorrect conclusion.

Prominent recent works in the field of distributed argumentation include: the argumentation context systems of [10], which, being abstract, do not specify how arguments are constructed or how the preference relation is built; and the distributed argumentation framework of [11], which is built on top of Defeasible Logic Programming, and does not integrate the notions of ambiguity propagating and team defeat.

CDL has already been deployed in Ambient Intelligence and Mobile Social Networks scenarios using logic programming algorithms and lightweight Prolog machines for mobile devices. To this direction, we plan to adjust the algorithms to support the new variants, but also develop more efficient algorithms, which will better suit the restricted computational capabilities of mobile devices. From a theoretical perspective, our plans include extending CDL with overlapping vocabularies and richer preference models.

References

1. Giunchiglia, F., Serafini, L.: Multilanguage Hierarchical Logics or: How we can do Without Modal Logics. Artificial Intelligence 65(1), 29–70 (1994)
2. Ghidini, C., Giunchiglia, F.: Local Models Semantics, or contextual reasoning = locality + compatibility. Artificial Intelligence 127(2), 221–259 (2001)
3. Roelofsen, F., Serafini, L.: Minimal and Absent Information in Contexts. In: IJCAI, pp. 558–563 (2005)
4. Brewka, G., Roelofsen, F., Serafini, L.: Contextual Default Reasoning. In: IJCAI, pp. 268–273 (2007)
5. Bikakis, A., Antoniou, G.: Contextual Argumentation in Ambient Intelligence. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 30–43. Springer, Heidelberg (2009)
6. Bikakis, A., Antoniou, G.: Defeasible Contextual Reasoning with Arguments in Ambient Intelligence. IEEE TKDE 22(11), 1492–1506 (2010)
7. Antoniou, G., Bikakis, A., Papatheodorou, C.: Reasoning with Imperfect Context and Preference Information in Multi-Context Systems. In: Catania, B., Ivanović, M., Thalheim, B. (eds.) ADBIS 2010. LNCS, vol. 6295, pp. 1–12. Springer, Heidelberg (2010)
8. Antoniou, G., Billington, D., Governatori, G., Maher, M.J., Rock, A.: A Family of Defeasible Reasoning Logics and its Implementation. In: ECAI, pp. 459–463 (2000)
9. Governatori, G., Maher, M.J., Antoniou, G., Billington, D.: Argumentation Semantics for Defeasible Logic. Journal of Logic and Computation 14(5), 675–702 (2004)
10. Brewka, G., Eiter, T.: Argumentation Context Systems: A Framework for Abstract Group Argumentation. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 44–57. Springer, Heidelberg (2009)
11. Thimm, M., Kern-Isberner, G.: A Distributed Argumentation Framework using Defeasible Logic Programming. In: COMMA, pp. 381–392. IOS Press, Amsterdam (2008)

On Influence and Contractions in Defeasible Logic Programming

Diego R. García[1], Sebastián Gottifredi[1], Patrick Krümpelmann[2], Matthias Thimm[2], Gabriele Kern-Isberner[2], Marcelo A. Falappa[1], and Alejandro J. García[1]

[1] Universidad Nacional del Sur, Bahía Blanca, Argentina
[2] Technische Universität Dortmund, Germany

Abstract. In this paper, we investigate the problem of contraction in Defeasible Logic Programming (DeLP), a logic-based approach for defeasible argumentation. We develop different notions of contraction based on both, the different forms of entailment implicitly existent in argumentation-based formalisms and the influence literals exhibit in the reasoning process. We give translations of widely accepted rationality postulates for belief contraction to our framework. Moreover we discuss on the applicability of contraction for defeasible argumentation and the role of influence in this matter.

1 Introduction

While most of the past work in argumentation has been done on the study of representation and inferential properties of different frameworks, the problem of belief change—one of the most important problems in knowledge representation—has not been investigated in depth so far, see [3] for a survey. Revision and the dynamics of beliefs in general have been studied for classical logics since the seminal paper [1]. There are also some proposals for dealing with belief change in non-classical logics, e. g. for defeasible logic in [2]. Here we consider the problem of contraction in the framework of *Defeasible Logic Programming* (DeLP) [4].

In DeLP, using a dialectical procedure involving arguments and counterarguments, literals can be established to be *warranted*, meaning that there are *considerable* grounds to believe the literal being true. The straightforward approach to define the success of contracting a defeasible logic program $\mathcal{P}$ by a literal l is to demand that l is not warranted anymore. This is similar to the approaches taken for classical theories [1] where success is defined in terms of non-entailment. In DeLP, however, there are three basic notions of "entailment": derivation (there are rules in $\mathcal{P}$ that allow to derive l), argument (there is an argument for l), and warrant (there is an undefeated argument for l). These notions lead to different alternatives of defining the success of a contraction. In addition to these notions of entailment, we also investigate the notion of *influence* in order to gain more insights into the problem of contraction. Due to the dialectical nature of the reasoning process employed in DeLP a literal can exhibit influence on the warrant status of other literals independently of its own warrant status. Contracting a program $\mathcal{P}$ by a literal l using the notion of warrant to define success still allows the dialectical procedure to use arguments for l in changing the warrant status of another literal.

This paper is organized as follows. In Section 2 we give a brief overview on the basic notions of Defeasible Logic Programming and continue with a thorough investigation of the notion of influence in Section 3. We apply the developed notions thereafter for investigating rationality postulates for contraction in our framework in Section 4. Finally, in Section 5 we present the conclusions of the work.

2 Defeasible Logic Programming

A single atom h or a negated atom $\sim h$ is called a literal or *fact*. Rules are divided into strict rules $h \leftarrow B$ and defeasible rules $h \prec B$ with a literal h and a set of literals B. A literal h is *derivable* from a set of facts and rules X, denoted by $X \mid\!\sim h$, iff it is derivable in the classical rule-based sense, treating strict and defeasible rules equally. A set X is *contradictory*, denoted $X \mid\!\sim \perp$, iff both $X \mid\!\sim h$ and $X \mid\!\sim \sim h$ holds for some h. A literal h is *consistently derivable* by X, denoted by $X \mid\!\sim^c h$, iff $X \mid\!\sim h$ and $X \not\mid\!\sim \perp$. A *defeasible logic program* (de.l.p.) $\mathcal{P}$ is a tuple $\mathcal{P} = (\Pi, \Delta)$ with a non-contradictory set of strict rules and facts Π and a set of defeasible rules Δ. We write $\mathcal{P} \mid\!\sim h$ as a shortcut for $\Pi \cup \Delta \mid\!\sim h$.

Definition 1 (Argument, Subargument). *Let h be a literal and let $\mathcal{P} = (\Pi, \Delta)$ be a de.l.p. Then $\langle \mathcal{A}, h\rangle$ with $\mathcal{A} \subseteq \Delta$ is an* argument *for h iff $\Pi \cup \mathcal{A} \mid\!\sim^c h$ and $\mathcal{A}$ is minimal wrt. set inclusion. A $\langle \mathcal{B}, q\rangle$ is a* subargument *of $\langle \mathcal{A}, h\rangle$ iff $\mathcal{B} \subseteq \mathcal{A}$.*

$\langle \mathcal{A}_1, h_1\rangle$ is a *counterargument* to $\langle \mathcal{A}_2, h_2\rangle$ at literal h, iff there is a subargument $\langle \mathcal{A}, h\rangle$ of $\langle \mathcal{A}_2, h_2\rangle$ such that $\Pi \cup \{h, h_1\}$ is contradictory.

In order to deal with counterarguments, a formal comparison criterion among arguments is used. Our results are independent of its choice, but as en example we use the *generalized specificity* relation $\succ$ [4]. Then, $\langle \mathcal{A}_1, h_1\rangle$ is a *defeater* of $\langle \mathcal{A}_2, h_2\rangle$, iff there is a subargument $\langle \mathcal{A}, h\rangle$ of $\langle \mathcal{A}_2, h_2\rangle$ such that $\langle \mathcal{A}_1, h_1\rangle$ is a counterargument of $\langle \mathcal{A}_2, h_2\rangle$ at literal h and either $\langle \mathcal{A}_1, h_1\rangle \succ \langle \mathcal{A}, h\rangle$ (*proper defeat*) or $\langle \mathcal{A}_1, h_1\rangle \not\succ \langle \mathcal{A}, h\rangle$ and $\langle \mathcal{A}, h\rangle \not\succ \langle \mathcal{A}_1, h_1\rangle$ (*blocking defeat*).

Definition 2 (Acceptable Argumentation Line). *Let $\Lambda = [\langle \mathcal{A}_1, h_1\rangle, \ldots, \langle \mathcal{A}_m, h_m\rangle]$ be a finite sequence of arguments. Λ is called an* acceptable argumentation line*, iff 1.) every $\langle \mathcal{A}_i, h_i\rangle$ with $i > 1$ is a defeater of $\langle \mathcal{A}_{i-1}, h_{i-1}\rangle$ and if $\langle \mathcal{A}_i, h_i\rangle$ is a blocking defeater of $\langle \mathcal{A}_{i-1}, h_{i-1}\rangle$ and $\langle \mathcal{A}_{i+1}, h_{i+1}\rangle$ exists, then $\langle \mathcal{A}_{i+1}, h_{i+1}\rangle$ is a proper defeater of $\langle \mathcal{A}_i, h_i\rangle$, 2.) $\Pi \cup \mathcal{A}_1 \cup \mathcal{A}_3 \cup \ldots$ is non-contradictory, 3.) $\Pi \cup \mathcal{A}_2 \cup \mathcal{A}_4 \cup \ldots$ is non-contradictory, and 4.) no $\langle \mathcal{A}_k, h_k\rangle$ is a subargument of some $\langle \mathcal{A}_i, h_i\rangle$ with $i < k$.*

In DeLP a literal h is *warranted*, if there is an argument $\langle \mathcal{A}, h\rangle$ which is non-defeated in the end. To decide whether $\langle \mathcal{A}, h\rangle$ is defeated or not, every acceptable argumentation line starting with $\langle \mathcal{A}, h\rangle$ has to be considered.

Definition 3 (Dialectical Tree). *Let $\mathcal{P} = (\Pi, \Delta)$ be a de.l.p. and let $\langle \mathcal{A}_0, h_0 \rangle$ be an argument. A* dialectical tree *for $\langle \mathcal{A}_0, h_0 \rangle$, denoted $\mathcal{T}\langle \mathcal{A}_0, h_0 \rangle$, is defined as follows: The root of $\mathcal{T}\langle \mathcal{A}_0, h_0 \rangle$ is $\langle \mathcal{A}_0, h_0 \rangle$. Let $\langle \mathcal{A}_n, h_n \rangle$ be a node in $\mathcal{T}\langle \mathcal{A}_0, h_0 \rangle$ and let $[\langle \mathcal{A}_0, h_0 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle]$ be a sequence of nodes. Let $\langle \mathcal{B}_1, q_1 \rangle, \ldots, \langle \mathcal{B}_k, h_k \rangle$ be the defeaters of $\langle \mathcal{A}_n, h_n \rangle$. For every defeater $\langle \mathcal{B}_i, q_i \rangle$ with $1 \leq i \leq k$ such that $[\langle \mathcal{A}_0, h_0 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle, \langle \mathcal{B}_i, q_i \rangle]$ is an acceptable argumentation line, the node $\langle \mathcal{A}_n, h_n \rangle$ has a child $\langle \mathcal{B}_i, q_i \rangle$. If there is no such $\langle \mathcal{B}_i, q_i \rangle$, $\langle \mathcal{A}_n, h_n \rangle$ is a leaf.*

In order to decide whether the argument at the root of a given dialectical tree is defeated or not, it is necessary to perform a *bottom-up*-analysis of the tree. Every leaf of the tree is marked "undefeated" and every inner node is marked "defeated", if it has at least one child node marked "undefeated". Otherwise it is marked "undefeated". Let $\mathcal{T}^*\langle \mathcal{A}_0, h_0 \rangle$ denote the marked dialectical tree of $\mathcal{T}\langle \mathcal{A}_0, h_0 \rangle$. We call a literal h *warranted* in a DeLP $\mathcal{P}$, denoted by $\mathcal{P} \mid\!\sim_w h$, iff there is an argument $\langle \mathcal{A}, h \rangle$ for h in $\mathcal{P}$ such that the root of the marked dialectical tree $\mathcal{T}^*\langle \mathcal{A}, h \rangle$ is marked "undefeated". Then $\langle \mathcal{A}, h \rangle$ is a *warrant* for h.

We will need some further notation in the following. Let $\mathcal{P} = (\Pi, \Delta)$ and $\mathcal{P}' = (\Pi', \Delta')$ be some programs and let r be a rule (either defeasible or strict). $\mathcal{P}$ is a subset of $\mathcal{P}'$, denoted by $\mathcal{P} \subseteq \mathcal{P}'$, iff $\Pi \subseteq \Pi'$ and $\Delta \subseteq \Delta'$. It is $r \in \mathcal{P}$ if either $r \in \Pi$ or $r \in \Delta$. We also define $\mathcal{P} \cup r =_{def} (\Pi, \Delta \cup \{r\})$ and $\mathcal{P} \cup \mathcal{A} =_{def} (\Pi, \Delta \cup \mathcal{A})$ for an argument $\mathcal{A}$.

3 Influence in Defeasible Logic Programs

It may be the case that an argument $\mathcal{A}$ for l is defeated in its own dialectical tree and not defeated in another tree [6]. Thus, these undefeated arguments for l may exhibit some *influence* one the marking status of arguments for another literal. Hence, by employing a contraction operation that bases its success only on the warrant status of the literal under consideration, it should be kept in mind that this literal might still have influence on the reasoning behavior in the contracted program. We therefore continue by investigating different notions of influence in more depth. Our first approach bases on an observation made when considering the removal of arguments for the literal l that has to be contracted.

Definition 4 (Trimmed Dialectical Tree). *Let $\mathcal{T}$ be some dialectical tree and l a literal. The l-*trimmed dialectical tree *$\mathcal{T} \setminus_t l$ is the same as $\mathcal{T}$ but every subtree $\mathcal{T}'$ of $\mathcal{T}$ with root $\langle \mathcal{A}_1, h \rangle$ and $\mathcal{A}_2 \subseteq \mathcal{A}_1$ such that $\langle \mathcal{A}_2, l \rangle$ is an argument for l is removed from $\mathcal{T}$.*

Note, that a trimmed dialectical tree is not a dialectical tree (as it is not complete) but that the marking procedure is still applicable in the same way.

Proposition 1. *Let l be a literal and $\mathcal{T}$ a dialectical tree. If $\mathcal{T} \setminus_t l$ is not empty and the marking of the root of $\mathcal{T}^*$ differs from the marking of the root of $(\mathcal{T} \setminus_t l)^*$ then there is an argument $\langle \mathcal{A}, k \rangle$ with $\mathcal{A}' \subseteq \mathcal{A}$ such that $\langle \mathcal{A}', l \rangle$ is an argument l and $\langle \mathcal{A}, k \rangle$ is undefeated in $\mathcal{T}^*$.*

Proposition 1 establishes that only an argument for l that is undefeated in $\mathcal{T}^*$ can possibly exhibit some influence. This leads to our first and most general definition of influence.

Definition 5 (Argument Influence $\mathcal{I}_\mathcal{A}$). *A literal l has* argument influence *in $\mathcal{P}$, denoted by $l \rightsquigarrow^\mathcal{A} \mathcal{P}$, if, and only if there is an argument $\langle \mathcal{A}_1, h\rangle$ with $\mathcal{A}_2 \subseteq \mathcal{A}_1$ such that $\langle \mathcal{A}_2, l\rangle$ is an argument for l and $\langle \mathcal{A}_1, h\rangle$ is a node in a dialectical tree $\mathcal{T}^*$ and $\langle \mathcal{A}_1, h\rangle$ is marked "undefeated" in $\mathcal{T}^*$.*

However, it is not the case that every argument that contains a subargument for l and is undefeated in some dialectical tree necessarily exhibits reasonable influence. This leads to our next notion of influence that only takes arguments into account which, on removal, will change the marking of the root.

Definition 6 (Tree Influence $\mathcal{I}_\mathcal{T}$). *A literal l has* tree influence *in $\mathcal{P}$, denoted by $l \rightsquigarrow^\mathcal{T} \mathcal{P}$, if and only if there is a dialectical tree $\mathcal{T}^*$ such that either 1.) the root's marking of $\mathcal{T}^*$ differs from the root's marking of $(\mathcal{T} \setminus_t l)^*$ or 2.) the root of $\mathcal{T}^*$ is marked "undefeated" and $(\mathcal{T} \setminus_t l)^*$ is empty.*

In order to establish whether a literal l exhibits *tree influence* every dialectical tree is considered separately. But recall, that for a literal h being warranted only the existence of a single undefeated argument is necessary. These considerations result in our final notion of *warrant influence.*

Definition 7 (Warrant Influence $\mathcal{I}_w$). *A literal l has* warrant influence *in $\mathcal{P}$, denoted by $l \rightsquigarrow^w \mathcal{P}$, if and only if there is a literal h such that either 1.) h is warranted in $\mathcal{P}$ and for every dialectical tree $\mathcal{T}^*$ rooted in an argument for h it holds that the root of $(\mathcal{T} \setminus_t l)^*$ is "defeated" or $(\mathcal{T} \setminus_t l)^*$ is empty, or 2.) h is not warranted in $\mathcal{P}$ and there is a dialectical tree $\mathcal{T}^*$ with the root being an argument for h and it holds that the root of $(\mathcal{T} \setminus_t l)^*$ is "undefeated".*

We conclude this section by providing a formal result that follows the iterative development of the notions of influences we gave above.

Proposition 2. *Given a de.l.p. $\mathcal{P}$ it holds that if $l \rightsquigarrow^w \mathcal{P}$ then $l \rightsquigarrow^\mathcal{T} \mathcal{P}$, and if $l \rightsquigarrow^\mathcal{T} \mathcal{P}$ then $l \rightsquigarrow^\mathcal{A} \mathcal{P}$.*

4 Rationality Postulates for Contraction in DeLP

The classic contraction operation $K - \phi$ for propositional logic is an operation that satisfies two fundamental properties, namely $\phi \notin Cn(K-\phi)$ and $K-\phi \subseteq K$. Hereby, the strong consequence operator of propositional logic and the resulting set of consequences $Cn(K)$ is the measure for the success of the contraction operation and therefore the scope of the considered effects of the operation. Given that we are dealing with a logic very different from propositional logic that is based on a dialectical evaluation of arguments we also have to consider a different *scope* on the effects for an appropriate contraction operation in this setting. For a de.l.p. $\mathcal{P}$ its *logical scope* is a set of literals that are relevant in a contraction scenario, i. e. that can be derived from $\mathcal{P}$ in some way, or that has some influence on $\mathcal{P}$.

Definition 8 (Logical Scope). *For $\mathcal{P}$ we define a class of* logical scopes $S_*(\mathcal{P})$ *via* $\mathcal{S}_d(\mathcal{P}) = \{l \mid P \mid\!\sim l\}$, $\mathcal{S}_w(\mathcal{P}) = \{l \mid P \mid\!\sim_w l\}$, $\mathcal{S}_{\mathcal{I}_\mathcal{A}}(\mathcal{P}) = \{l \mid l \rightsquigarrow^\mathcal{A} \mathcal{P}\}$, $\mathcal{S}_{\mathcal{I}_\mathcal{T}}(\mathcal{P}) = \{l \mid l \rightsquigarrow^\mathcal{T} \mathcal{P}\}$, *and* $\mathcal{S}_{\mathcal{I}_w}(\mathcal{P}) = \{l \mid l \rightsquigarrow^w \mathcal{P}\}$.

Proposition 3. *For any d.e.lp. $\mathcal{P}$ it holds that $\mathcal{S}_w(\mathcal{P}) \subseteq \mathcal{S}_{\mathcal{I}_w}(\mathcal{P}) \subseteq \mathcal{S}_{\mathcal{I}_\mathcal{T}}(\mathcal{P}) \subseteq \mathcal{S}_{\mathcal{I}_\mathcal{A}}(\mathcal{P}) \subseteq \mathcal{S}_d(\mathcal{P})$.*

Based on the notion of scopes we propose specifications of contraction operators with different scopes by sets of postulates that resemble the rationality postulates for contraction in classic belief change theory.

Definition 9. *For a de.l.p. $\mathcal{P}$ and a literal l, let $\mathcal{P} - l = (\Pi', \Delta')$ be the result of contracting $\mathcal{P}$ by l. We define the following set of postulates for different scopes $*$ with $* \in \{d, w, \mathcal{I}_\mathcal{T}, \mathcal{I}_\mathcal{A}, \mathcal{I}_w\}$.*
(Success$_*$) $l \notin \mathcal{S}_*(\mathcal{P} - l)$
(Inclusion) $\mathcal{P} - l \subseteq \mathcal{P}$
(Vacuity$_*$) *If $l \notin \mathcal{S}_*(\mathcal{P})$, then $\mathcal{P} - l = \mathcal{P}$*
(Core-retainment$_*$) *If $k \in \mathcal{P}$ (either a fact or a rule) and $k \notin \mathcal{P} - l$, then there is a de.l.p. $\mathcal{P}'$ such that $\mathcal{P}' \subseteq \mathcal{P}$ and such that $l \notin \mathcal{S}_*(\mathcal{P}')$ but $l \in \mathcal{S}_*(\mathcal{P}' \cup \{k\})$*

These postulates represent the adaptation of applicable postulates from belief base contraction [5] to de.l.p. program contraction. The first postulate defines when the contraction is considered successful, which in our case is dependent on the scope of the contraction. The second postulate states that we are actually contracting the belief base. The third postulate requires that if the literal to be contracted is out of the scope of the operator then nothing should be changed, while the fourth postulate states that only relevant facts or rules should be erased and hence demands for minimality of change.

Definition 10. *$P - l$ is called a $*$-contraction if and only if it satisfies (Success$_*$), (Inclusion), (Vacuity$_*$) and (Core-retainment$_*$) with $* \in \{d, w, \mathcal{I}_\mathcal{T}, \mathcal{I}_\mathcal{A}, \mathcal{I}_w\}$.*

In the following we are investigating constructive approaches based on kernel sets for defining $*$-contraction operations (with $* \in \{d, w, \mathcal{I}_\mathcal{T}, \mathcal{I}_\mathcal{A}, \mathcal{I}_w\}$).

Definition 11. *Let $\mathcal{P} = (\Pi, \Delta)$ be a de.l.p. and let α be a literal. An α-*kernel *H of $\mathcal{P}$ is a set $H = \Pi' \cup \Delta'$ with $\Pi' \subseteq \Pi$ and $\Delta' \subseteq \Delta$ such that 1.) $(\Pi', \Delta') \mid\!\sim \alpha$, 2.) $(\Pi', \Delta') \not\mid\!\sim \perp$, and H is minimal wrt. set inclusion. The set of all α-kernels of $\mathcal{P}$ is called the* kernel set *and is denoted by $\mathcal{P} \perp\!\!\!\perp \alpha$.*

Note that the notion of an α-kernel is not equivalent to the notion of an argument as an argument consists only of a set of defeasible rules while an α-kernel consists of all rules needed to derive α, in particular strict rules and facts.

As in [5] we define contractions in terms of *incision functions*. A function σ is called an *incision function* if *(1)* $\sigma(\mathcal{P} \perp\!\!\!\perp \alpha) \subseteq \bigcup \mathcal{P} \perp\!\!\!\perp \alpha$ and *(2)* $\emptyset \subset H \in \mathcal{P} \perp\!\!\!\perp \alpha$ implies $H \cap \sigma(\mathcal{P} \perp\!\!\!\perp \alpha) \neq \emptyset$. This general definition for an incision function removes at least one element in every kernel set thus inhibiting every derivation of α. Such an incision function is adequate for realizing a d-contraction but it is too strict for our more general notions of contraction. We therefore drop the second condition above for incision functions used in this work.

Definition 12. *Let $\mathcal{P}$ be a de.l.p., let α be a literal, and let $\mathcal{P}\perp\!\!\!\perp\alpha$ be the kernel set of $\mathcal{P}$ with respect to α. A function σ is a* dialectical incision function *iff $\sigma(\mathcal{P}\perp\!\!\!\perp\alpha) \subseteq \bigcup \mathcal{P}\perp\!\!\!\perp\alpha$.*

Using dialectical incision functions we can define a contraction operation in DeLP as follows.

Definition 13. *Let σ be a dialectical incision function for $\mathcal{P} = (\Pi, \Delta)$. The* dialectical kernel contraction $-_\sigma$ *for $\mathcal{P}$ is defined as $\mathcal{P}-_\sigma\alpha = (\Pi \setminus \sigma(\mathcal{P}\perp\!\!\!\perp\alpha), \Delta \setminus \sigma(\mathcal{P}\perp\!\!\!\perp\alpha))$. Conversely, a contraction operator $\div$ for $\mathcal{P}$ is called a* dialectical kernel contraction *if and only if there is some dialectical incision function σ for $\mathcal{P}$ such that $\mathcal{P} \div \alpha = \mathcal{P}-_\sigma\alpha$ for all literals α.*

Due to lack of space we give only the definition for *$\mathcal{I}_\mathcal{T}$-incision function*, the other incision functions are defined analogously.

Definition 14. *Let (Π, Δ) be a de.l.p., α a literal, and σ be a dialectical incision function with $\sigma((\Pi, \Delta)\perp\!\!\!\perp\alpha) = S$. Then σ is an $\mathcal{I}_\mathcal{T}$-*incision function *if 1.) $\alpha \not\mathrel{|\!\!\sim}^{\mathcal{T}} (\Pi \setminus S, \Delta \setminus S)$ and 2.) there is no $S' \subset S$, such that S' satisfies 1.).*

Proposition 4. *Let (Π, Δ) be a de.l.p., α a literal, and let $* \in \{d, \mathcal{I}_\mathcal{A}, \mathcal{I}_\mathcal{T}, \mathcal{I}_w, w\}$. If σ is a dialectical $*$-incision function then $-_\sigma$ is a $*$-contraction.*

5 Conclusions

In this work we started the investigation of contraction operations in defeasible logic programs. We identified different approaches to contraction depending on the notion of success of the operation. Besides the notions based on entailment and warrant we elaborated on more fine grained differences based on notions of influence. This lead to the definition of rationality postulates for each type of contraction. Furthermore, we showed that each contraction operation is constructible using kernel sets in the style of classic belief contraction.

References

1. Alchourrón, C.E., Gärdenfors, P., Makinson, D.: On the logic of theory change: Partial meet contraction and revision functions. Journal of Symbolic Logic 50(2), 510–530 (1985)
2. Billington, D., Antoniou, G., Governatori, G., Maher, M.J.: Revising nonmonotonic theories: The case of defeasible logic. In: Burgard, W., Christaller, T., Cremers, A.B. (eds.) KI 1999. LNCS (LNAI), vol. 1701, pp. 101–112. Springer, Heidelberg (1999)
3. Falappa, M.A., Kern-Isberner, G., Simari, G.R.: Belief revision and argumentation theory. In: Argumentation in Artificial Intelligence, pp. 341–360. Springer, Heidelberg (2009)
4. Garcia, A., Simari, G.R.: Defeasible logic programming: An argumentative approach. Theory and Practice of Logic Programming 4(1-2), 95–138 (2004)
5. Hansson, S.O.: Kernel contraction. J. of Symbolic Logic 59, 845–859 (1994)
6. Thimm, M., Kern-Isberner, G.: On the relationship of defeasible argumentation and answer set programming. In: COMMA 2008, pp. 393–404 (2008)

Termination of Grounding Is Not Preserved by Strongly Equivalent Transformations

Yuliya Lierler[1] and Vladimir Lifschitz[2]

[1] University of Kentucky
yuliya@cs.uky.edu
[2] University of Texas at Austin
vl@cs.utexas.edu

Abstract. The operation of a typical answer set solver begins with grounding—replacing the given program with a program without variables that has the same answer sets. When the given program contains function symbols, the process of grounding may not terminate. In this note we give an example of a pair of consistent, strongly equivalent programs such that one of them can be grounded by LPARSE, DLV, and GRINGO, and the other cannot.

1 Introduction

The operation of a typical answer set solver, such as SMODELS[1], DLV[2], or CLINGO[3], begins with "intelligent instantiation," or grounding—replacing the given program with a program without variables that has the same answer sets. When the given program contains function symbols, the process of grounding may not terminate. The grounder employed in the last (2010-10-14) release of DLV terminates when the input program is finitely ground in the sense of [1]. According to Theorem 5 from that paper, the class of finitely ground programs is undecidable. Before attempting to ground a program, DLV verifies a decidable condition that guarantees the termination of grounding. (Conditions of this kind are known, for instance, from [1, Section 5] and [2].) A command-line option can be used to override this "finite domain check"; ensuring termination becomes then the responsibility of the user.

In the course of a public discussion at a recent meeting[4], Michael Gelfond observed that the behavior of the current version of DLV may not be fully declarative, because of the possibility of nontermination, in the same sense in which the behavior of standard Prolog systems is not fully declarative: an "inessential" modification of a Prolog program may affect not only its runtime but even its termination, even the possibility of getting an output in principle.

[1] http://www.tcs.hut.fi/Software/smodels/
[2] http://www.dlvsystem.com/
[3] http://potassco/sourceforge.net/
[4] NonMon@30: Thirty Years of Nonmonotonic Reasoning, Lexington, KY, October 22–25, 2010.

It is well known that termination of Prolog can be affected by very minor changes, such as changing the order of subgoals in the body of a rule, changing the order of rules, or inserting trivial rules of the form $A \leftarrow A$. In the case of DLV, such modifications cannot affect termination. But isn't it possible that a slightly more complex transformation that has no effect on the meaning of the program would make the DLV grounder unusable?

Our goal in this note is to investigate to what degree this suspicion is justified. To make Gelfond's question precise, we need to explain what we mean by a transformation that has no effect on the meaning of the program. One possibility is consider transformations that are strongly equivalent in the sense of [3,4]. Recall that logic programs Π_1 and Π_2 are said to be *strongly equivalent* to each other if, for every logic program Π, programs $\Pi \cup \Pi_1$ and $\Pi \cup \Pi_2$ have the same answer sets. For instance, changing the order of subgoals in the body of a rule produces a strongly equivalent program. The same can be said about changing the order of rules and about inserting a rule of the form $A \leftarrow A$. Further examples of strongly equivalent transformations are provided by removing rules that are "subsumed" by other rules of the program. For instance, a program of the form

$$\begin{array}{l} A \leftarrow B \\ A \leftarrow B, C \end{array}$$

is strongly equivalent to its first rule $A \leftarrow B$.

In this note, we give an example of a pair of consistent, strongly equivalent programs Π_1 and Π_2 such that Π_1 is finitely ground, and Π_2 is not. Thus one of these two "essentially identical" programs can be grounded by DLV, and the other cannot. The behavior of LPARSE (the grounder of SMODELS) and GRINGO 2.0.3 (the grounder of the latest version of CLINGO) is similar: they terminate on Π_1, but not on Π_2.

2 The Example

Program Π_1 consists of 4 rules:

$$\begin{array}{l} p(a) \\ q(X) \leftarrow p(X) \\ \leftarrow p(f(X)), q(X) \\ r(f(X)) \leftarrow q(X),\ not\ p(f(X)). \end{array}$$

According to the answer set semantics [5], Π_1 is shorthand for the set of ground instances of its rules:

$$\begin{array}{l} p(a) \\ q(f^i(a)) \leftarrow p(f^i(a)) \\ \leftarrow p(f^{i+1}(a)), q(f^i(a)) \\ r(f^{i+1}(a)) \leftarrow q(f^i(a)),\ not\ p(f^{i+1}(a)) \end{array}$$

$(i = 0, 1, \dots)$. It is easy to see that

$$\{p(a), q(a), r(f(a))\} \tag{1}$$

is an answer set of Π_1. Indeed, the reduct of Π_1 relative to this set is

$$\begin{array}{l} p(a) \\ q(f^i(a)) \leftarrow p(f^i(a)) \\ \leftarrow p(f^{i+1}(a)), q(f^i(a)) \\ r(f^{i+1}(a)) \leftarrow q(f^i(a)) \end{array}$$

$(i = 0, 1, \dots)$, and (1) is a minimal set of ground atoms satisfying these rules. As a matter of fact, each of the three grounders discussed in this note turns Π_1 into the set of facts (1) and tells us in this way that (1) is the *only* answer set of Π_1.

Program Π_2 is obtained from Π_1 by adding the rule

$$p(f(X)) \leftarrow q(X),\ \mathit{not}\ r(f(X)). \tag{2}$$

We will show that programs Π_1 and Π_2 are strongly equivalent to each other. In fact, this claim will remain true even if we drop the first two rules from each of the programs:

Proposition 1. *The program*

$$\begin{array}{l} \leftarrow p(f(X)), q(X) \\ r(f(X)) \leftarrow q(X),\ \mathit{not}\ p(f(X)) \end{array} \tag{3}$$

is strongly equivalent to

$$\begin{array}{l} \leftarrow p(f(X)), q(X) \\ r(f(X)) \leftarrow q(X),\ \mathit{not}\ p(f(X)) \\ p(f(X)) \leftarrow q(X),\ \mathit{not}\ r(f(X)). \end{array} \tag{4}$$

In other words, if we take any program containing rules (3) and add to it the last of rules (4) then the answer sets of the program will remain the same.

Second, we will show that Π_1 is finitely ground, and Π_2 is not. In fact, Π_1 belongs to the class of finite domain programs—the decidable set of finitely ground programs introduced in [1].

Proposition 2. *Π_1 is a finite domain program.*

Proposition 3. *Program Π_2 is not finitely ground.*

3 Proofs

3.1 Proof of Proposition 1

In view of the main theorem of [4], it is sufficient to prove the following fact:

Lemma. *The formula*

$$q(x) \wedge \neg r(f(x)) \rightarrow p(f(x)) \tag{5}$$

can be derived from the formulas

$$\begin{array}{l}\neg(p(f(x)) \wedge q(x)),\\ q(x) \wedge \neg p(f(x)) \rightarrow r(f(x))\end{array} \tag{6}$$

in propositional intuitionistic logic.

Proof of the Lemma. The formula

$$\neg(q(x) \wedge \neg r(f(x))) \tag{7}$$

can be derived from (6) in classical propositional logic. By Glivenko's theorem[5], it follows that it can be derived from (6) intuitionistically as well. It remains to observe that (5) is an intuitionistic consequence of (7).

3.2 Proof of Proposition 2

In this section, and in the proof of Proposition 3 below as well, we assume that the reader is familiar with the terminology and notation introduced in [1].

To show that Π_1 is a finite domain program we need to check that the arguments $p[1]$, $q[1]$, $r[1]$ of the predicates of Π_1 are finite-domain arguments [1, Definition 10]. Consider the argument $p[1]$. The only rule of Π_1 with p in the head is $p(a)$. This rule satisfies Condition 1 from Definition 10. Consider the argument $q[1]$. The only rule with q in the head is

$$q(X) \leftarrow p(X).$$

This rule satisfies Condition 2. Consider the argument $r[1]$. The only rule with r in the head is

$$r(f(X)) \leftarrow q(X),\ \mathit{not}\ p(f(X)).$$

This rule satisfies Condition 3.

3.3 Proof of Proposition 3

To prove that program Π_2 is not finitely ground we need to find a component ordering ν for Π_2 such that the intelligent instantiation of Π_2 for ν is infinite [1, Definition 9]. The only component ordering for Π_2 is

$$\langle C_{\{p,q\}}, C_{\{r\}} \rangle,$$

as can be seen from the dependency graph $\mathcal{G}(\Pi_2)$ and the component graph $\mathcal{G}^C(\Pi_2)$ of this program [1, Definitions 1–4]; these graphs are shown in Figure 1. According to [1, Definition 8], the intelligent instantiation of Π_2 for this component ordering is the set S_2 of ground rules defined by the formulas

$$\begin{array}{l}S_1 = \Phi^{\infty}_{\Pi_2(C_{\{p,q\}}),S_0}(\emptyset),\\ S_2 = S_1 \cup \Phi^{\infty}_{\Pi_2(C_{\{r\}}),S_1}(\emptyset).\end{array}$$

[5] This theorem [6], [7, Theorem 3.1] asserts that if a formula beginning with negation can be derived from a set Γ of formulas in classical propositional logic then it can be derived from Γ in intuitionistic propositional logic as well.

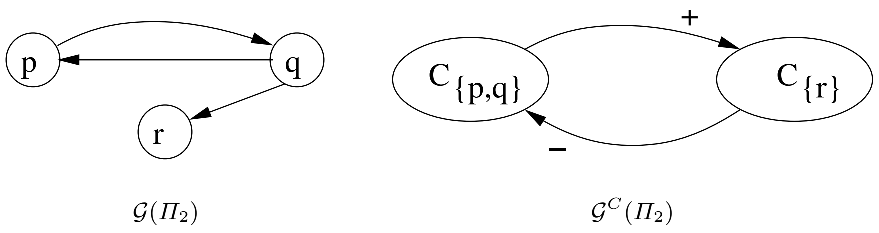


Fig. 1. The dependency graph and the component graph of program Π_2

The module $\Pi_2(C_{\{p,q\}})$ is the program

$$\begin{array}{l} p(a) \\ q(X) \leftarrow p(X) \\ p(f(X)) \leftarrow q(X),\ not\ r(f(X)), \end{array}$$

and the k-th iteration of the operator $\Phi_{\Pi_2(C_{\{p,q\}}),S_0}$ on the empty set, for $k \geq 1$, consists of the rules

$$\begin{array}{ll} p(a), & \\ & \\ q(f^j(a)) \leftarrow p(f^j(a)) & (0 \leq j \leq \frac{k}{2} - 1), \\ & \\ p(f^{j+1}(a)) \leftarrow q(f^j(a)),\ not\ r(f^{j+1}(a)) & (0 \leq j \leq \frac{k-3}{2}). \end{array}$$

It is clear that the union S_1 of these iterations is infinite, and so is S_2.

4 Conclusion

Our goal was to find two nontrivial programs that have essentially the same meaning (which we chose to understand as "consistent programs that are strongly equivalent to each other") such that one of them can be grounded, and the other cannot. The pair Π_1, Π_2 is the simplest example that we could come up with, and the claim that these programs have essentially the same meaning is far from obvious (recall the proof of the lemma in Section 3.1). So the view that the possibility of nontermination makes the behavior of answer set solvers nondeclarative may not be justified, after all.

The fact that the termination of grounding is not preserved by strongly equivalent transformations shows, on the other hand, that such a transformation may serve as a useful preprocessing step before an attempt to ground a program.

Acknowledgements

Thanks to Michael Gelfond and Wolfgang Faber for useful discussions. Yuliya Lierler was supported by a 2010 Computing Innovation Fellowship. Vladimir Lifschitz was partially supported by the National Science Foundation under Grant IIS-0712113.

References

1. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 407–424. Springer, Heidelberg (2008)
2. Lierler, Y., Lifschitz, V.: One more decidable class of finitely ground programs. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 489–493. Springer, Heidelberg (2009)
3. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2, 526–541 (2001)
4. Lifschitz, V., Pearce, D., Valverde, A.: A characterization of strong equivalence for logic programs with variables. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 188–200. Springer, Heidelberg (2007)
5. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
6. Glivenko, V.: Sur quelques points de la logique de M. Brouwer. Académie Royale de Belgique. Bulletins de la Classe des Sciences, se'rie 5 15, 183–188 (1929)
7. Mints, G.: A Short Introduction to Intuitionistic Logic. Kluwer, Dordrecht (2000)

Aggregates in Answer Set Optimization

Emad Saad[1] and Gerhard Brewka[2]

[1] Gulf University for Science and Technology, Mishref, Kuwait
saad.e@gust.edu.kw
[2] University of Leipzig, Augustusplatz 10-11, D-04109 Leipzig
brewka@informatik.uni-leipzig.de

Abstract. Answer set optimization (ASO) is a flexible framework for qualitative optimization in answer set programming (ASP). The approach uses a generating program to construct the space of problem solutions, and a preference program to assess the quality of solutions. In this short paper we generalize the approach by introducing aggregates in preference programs. This allows the user to express preferences which are based on minimization or maximization of some numerical criteria. We introduce the new language of preference programs, define its semantics and give an example illustrating its use.

1 Introduction

Preferences were introduced in answer set programming in various forms. In [8], preferences are defined among the rules of the logic program, whereas preferences among literals are investigated in [7]. Logic programs with ordered disjunction (LPOD) [2] represent context-dependent preferences using a specific connective in the heads of rules. The ASO approch [1] separates answer set generation from evaluation, using so-called preference programs for the latter.

On the other hand, the lack of aggregate preferences, e.g., minimum and maximum, in ASO and LPOD makes them less suitable for intuitive and easy encoding of some real-world applications like, for example, finding Nash equilibria in strategic games and general optimization problems. Consider the prisoner dilemma strategic game encoding in ASO or LPOD as presented in [5]. The dilemma is described as follows:

Two criminals are held in separate cells for committing a crime. They are not allowed to communicate. If both criminals keep quiet, do not fink on each other, then each will be convicted in part for a crime and spend one year in prison. However, if both fink, then each one will spend three years in prison. But, if one finks and the other keeps quiet, the one who finks will be freed and the other will spend four years in prison.

The payoff function for the first criminal, u_1, assigns 3 to $(fink, quiet)$, which means that the first criminal gets a payoff of 3 if (s)he finks and the second criminal keeps quiet. Similarly, $u_1(quiet, quiet) = 2, u_1(fink, fink) = 1$, and $u_1(quiet, fink) = 0$. In the same way, the payoff function for the second criminal, u_2, is defined as $u_2(fink, quiet) = 0$, which means that the second criminal gets a payoff of 0 if the first criminal finks and the (s)he keeps quiet. Similarly, $u_2(quiet, quiet) = 2, u_2(fink, fink) = 1$, and $u_2(quiet, fink) = 3$.

This prisoner dilemma strategic game has a single Nash equilibrium, namely $(fink, fink)$. The game can be represented by an ASO program $P = \langle P_{gen}, P_{pref} \rangle$, where P_{gen} is a disjunctive logic program and contains the rules:

$$\begin{array}{l} fink_1 \vee quiet_1 \leftarrow \\ fink_2 \vee quiet_2 \leftarrow \end{array}$$

and P_{pref} consists of the preference rules:

$$\begin{array}{l} fink_1 > quiet_1 \leftarrow quiet_2 \\ fink_1 > quiet_1 \leftarrow fink_2 \\ fink_2 > quiet_2 \leftarrow quiet_1 \\ fink_2 > quiet_2 \leftarrow fink_1 \end{array}$$

where $fink_i$ or $quiet_i$, for $i \in \{1, 2\}$, denotes player i chooses to fink or to be quiet, respectively. According to the semantics of ASO, the above program has the four answer sets $\{fink_1, fink_2\}, \{fink_1, quiet_2\}, \{quiet_1, fink_2\}$, and $\{quiet_1, quiet_2\}$. According to P_{pref} $\{fink_1, fink_2\}$ is the most preferred answer set, which coincides with the Nash equilibrium of the game.

To find the answer sets that coincide with the Nash equilibria of the game, a natural ASO program encoding of the game should be able to encode the payoff functions of the players along with preference relations that maximize the players payoffs. The current syntax and semantics of ASO do not define preference relations or rank answer sets based on minimization or maximization of some desired criterion specified by the user. In addition, in order to provide a correct ASO encoding and solution to the game, the user has to examine all the players' payoff functions to find out each player's best actions in response to the other players' actions. This can be infeasible for large games with large number of actions and players. This makes ASO approach to finding Nash equilibria is not fully automated, as the user participates in the solution of the problem.

The same also applies when we try to encode general optimization problems in ASO, where the most preferred solutions are the ones that maximize or minimize some utility function specified by the user. We call preference relations that are used to maximize or minimize some desired criterion aggregate preferences. For these reasons, we extend ASO programs with aggregate preferences.

2 ASOG Programs: Syntax

An ASOG (answer set optimization with aggregate preferences) program is a pair of programs $P = \langle P_{gen}, P_{pref} \rangle$. As in the ASO approach, P_{gen} is an arbitrary logic program generating answer sets. P_{pref}, called aggregate preference program, is a set of rules that represent the user preferences. The preferences are used to rank the generated answer sets. We first introduce the syntax of aggregate preference programs.

Our treatment of aggregates follows [3]. Let L be a first-order language with finitely many predicate symbols, function symbols, constants, and infinitely many variables. A standard literal is either a standard atom a in B_L or the negation of a ($\neg a$), where B_L is the Herbrand base of L and $\neg$ is the classical negation. The Herbrand universe of L is denoted by U_L. Non-monotonic negation or the negation as failure is denoted by *not*. Let *Lit* be the set of all standard literals in L, where $Lit = \{a | a \in B_L\} \cup \{\neg a | a \in B_L\}$.

A pair of the form $\{V : C\}$, where V is a list of variables and C is a conjunction of standard literals from *Lit*, is called a symbolic set. A ground set is a set of pairs $\{\langle \overline{Const} : C \rangle\}$ where $\overline{Const}$ is a list of constants and C is a ground conjunction of standard literals. A ground set or a symbolic set is called a set term. We say $f(S)$ is an aggregate function if S is a set term and f is one of the aggregate function symbols in $\{min, max, count, sum, times\}$. If $f(S)$ is an aggregate function, T a constant or variable term and $\circ \in \{=, \neq, <, >, \leq, \geq\}$, then we say $f(S) \circ T$ is an aggregate atom with guard T. An optimization aggregate is an expression of the form $max(f(S))$ or $min(f(S))$, where S is a set term and f is an aggregate function symbol.

A variable appearing solely in a set term is called a local variable, otherwise it is a global variable. Let A be a set of standard literals, aggregate atoms, and optimization aggregates. A boolean combination over A is a boolean formula over standard literals, aggregates atoms, optimization aggregates in A constructed by conjunction, disjunction, classical negation ($\neg$), and non-monotonic negation (*not*), where classical negation is combined only with standard atoms (to form standard literals) and non-monotonic negation is combined only with standard literals and aggregate atoms.

Definition 1. *A preference program, P_{pref}, over a set of standard literals, aggregate atoms, and optimization aggregates, A, is a finite set of preference rules of the form*

$$C_1 > C_2 > \ldots > C_k \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_{m+n} \tag{1}$$

where $l_1, \ldots, l_{n+m}$ are standard literals or aggregate atoms and $C_1, C_2, \ldots, C_k$ are boolean combinations over A.

Intuitively, any answer set that satisfies $body = l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_{m+n}$ and C_1 is preferred over answer sets that satisfy *body*, some C_i ($2 \leq i \leq k$), but not C_1, and any answer set that satisfies *body* and C_2 is preferred over answer sets that satisfy *body*, some C_i ($3 \leq i \leq k$), but neither C_1 nor C_2, etc.

Definition 2. *An ASOG program, is a pair of programs $\langle P_{gen}, P_{pref} \rangle$, where P_{gen} is a logic program with answer set semantics, called the generating program, and P_{pref} is a preference program.*

3 ASOG Programs: Semantics

A global substitution for a preference rule, r, is a substitution of the set of global variables in r to constants from U_L. A local substitution of a symbolic set S is a substitution of the set of local variables appearing in S to constants from U_L. Let $S = \{V : C\}$ be a symbolic set with no global variables. The instantiation of S, denoted by $inst(S)$, is a set of ground pairs $inst(S) = \{\langle\theta\,(V)\ :\ \theta\,(C)\rangle \mid \theta$ is a local substitution for S $\}$. A ground instance of a preference rule r is given as follows. Given a global substitution γ, first $\gamma(r)$, the instantiation of r by γ, is found, then $inst(S)$ replaces every symbolic set S in $\gamma(r)$. The ground instantiation of a preference program, P_{pref}, denoted by $Ground(P_{pref})$, is the set of all possible ground instances of every rule, r, in P_{pref}. The semantics of the aggregate functions $min, max, count, sum$, and $times$ are defined by appropriate mappings, where the types of the functions are as follows: $min, max : (\overline{2}^{\mathbb{N}} - \emptyset) \rightarrow \mathbb{I}$; $count : (\overline{2}^{U_L} - \emptyset) \rightarrow \mathbb{I}$, $sum : \overline{2}^{\mathbb{N}} \rightarrow \mathbb{I}$, and $times : \overline{2}^{\mathbb{N}^+} \rightarrow \mathbb{I}$, where $\mathbb{I}, \mathbb{N}, \mathbb{N}^+$ is the set of all integers, natural numbers, and non-negative numbers, respectively, and for a set X, $\overline{2}^{\mathbb{X}}$ is the set of all multisets over elements in X. The application of sum and $times$ on the empty multiset returns 0 and 1, respectively. However, the application of max, min, and $count$ on the empty multiset is undefined. Let $\perp$ be a symbol that does not occur in an ASOG program P.

Definition 3. *Let P be a generating program, l a standard ground literal, $f(S) \circ T$ a ground aggregate atom, I a set of ground standard literals. Let A_I^S be the multiset $\{\!\{t_1 | \langle t_1, t_2, \ldots, t_n : Conj\rangle \in S \wedge$ Conj holds in $I\}\!\}$. Let $f(A_I^S) = \perp$ if A_I^S is not in the domain of f. The satisfaction of a boolean combination, C, by a set of ground standard literals I under P, denoted by $I \models^P C$, is defined inductively as follows:*

- *$I \models^P l$ iff $l \in I$.*
- *$I \models^P$ not l iff $l \notin I$.*
- *$I \models^P f(S) \circ T$ iff $f(A_I^S) \circ T$ and $f(A_I^S) \neq \perp$.*
- *$I \models^P$ not $f(S) \circ T$ iff $f(A_I^S) \not\circ T$ and $f(A_I^S) \neq \perp$.*
- *$I \models^P max(f(S))$ iff $f(A_I^S) \neq \perp$ and for any answer set I' of P, $f(A_{I'}^S) \neq \perp$ implies $f(A_{I'}^S) \leq f(A_I^S)$.*
- *$I \models^P min(f(S))$ iff $f(A_I^S) \neq \perp$ and for any answer set I' of P, $f(A_{I'}^S) \neq \perp$ implies $f(A_I^S) \leq f(A_{I'}^S)$.*
- *$I \models^P C_1 \wedge C_2$ iff $I \models^P C_1$ and $S \models^P C_2$.*
- *$I \models^P C_1 \vee C_2$ iff $I \models^P C_1$ or $I \models^P C_2$.*

Note that P is only required for $max(f(S))$ and $min(f(S))$, where all answer sets of P need to be examined. When P is irrelevant or clear from context we will often leave out the upper index and simply write $\models$.

The application of any aggregate function except $count$, f, on a singleton $\{a\}$, returns a, i.e., $f(\{a\}) = a$. Therefore, we use $max(S)$ and $min(S)$ as abbreviations for the optimization aggregates $max(f(S))$ and $min(f(S))$ respectively, where S is a singleton and f is arbitrary aggregate function except $count$.

The following definition specifies the satisfaction of the preference rules.

Definition 4. *Let r be a preference rule of the form (1) and I be a set of ground standard literals. We define the following types of satisfaction of r under I:*

– *$I \models_i r$ iff I satisfies the body of r as well as C_i, and $i = \min\{l \mid I \models C_l\}$.*
– *$I \models_{irr} r$ iff either I does not satisfy the body of r or I satisfies the body of r but none of the C_i in the head of r*[1].

Definition 5. *Let r be a preference rule of the form (1) and I_1, I_2 sets of ground standard literals. Then, I_1 is at least as preferred as I_2 w.r.t. r, denoted by $I_1 \geq_r I_2$, iff one of the following holds:*

– *$I_1 \models_i r$, $I_2 \models_j r$ and $i \leq j$.*
– *$I_2 \models_{irr} r$.*

We say I_1 is strictly preferred to I_2, denoted $I_1 >_r I_2$, iff $I_1 \geq_r I_2$ but not $I_2 \geq_r I_1$. I_1 and I_2 are equally preferred w.r.t. r, denoted $I_2 =_r I_1$, iff $I_1 \geq_r I_2$ and $I_2 \geq_r I_1$.

Definition 5 specifies the ordering of answer sets according to a single preference rule[2]. The following definitions determine the ordering of answer sets with respect to preference programs.

Definition 6 (Pareto preference). *Let $P = \langle P_{gen}, P_{pref} \rangle$ be an ASOG program and let S_1, S_2 be two answer sets of P_{gen}. Then, S_1 is (Pareto) preferred over S_2 w.r.t. P_{pref}, denoted by $S_1 >_{P_{pref}} S_2$, iff there is at least one preference rule $r \in P_{pref}$ such that $S_1 >_r S_2$ and for every other rule $r' \in P_{pref}$, $S_1 \geq_{r'} S_2$. We say, S_1 and S_2 are equally (Pareto) preferred w.r.t. P_{pref}, denoted by $S_1 =_{P_{pref}} S_2$, iff for all $r \in P_{pref}$, $S_1 =_r S_2$.*

Definition 7 (Cardinality preference). *Let $P = \langle P_{gen}, P_{pref} \rangle$ be an ASOG program. The relation $\succeq_c$ on the answer sets of P_{gen} is the smallest transitive relation containing (S_1, S_2) whenever*

$$|\{r \in P_{pref} | S_1 \geq_r S_2\}| > |\{r \in P_{pref} | S_2 \geq_r S_1\}|.$$

S_1 is called maximally cardinality preferred iff for each answer set S_2 of P_{gen} we have: $S_2 \succeq_c S_1$ implies $S_1 \succeq_c S_2$[3].

Example. We now present an ASOG encoding of the prisoner dilemma strategic game presented in the introduction. It is convenient to use cardinality constraints in P_{gen}:

$$\begin{array}{cc}
action_1(quiet_1) & action_1(fink_1) \\
action_2(quiet_2) & action_2(fink_2) \\
1\{choose(A_1) : action_1(A_1)\}1 & 1\{choose(A_2) : action_2(A_2)\}1 \\
u(quiet_1, quiet_2, 2, 2) & \leftarrow choose(quiet_1), choose(quiet_2) \\
u(quiet_1, fink_2, 0, 3) & \leftarrow choose(quiet_1), choose(fink_2) \\
u(fink_1, quiet_2, 3, 0) & \leftarrow choose(fink_1), choose(quiet_2) \\
u(fink_1, fink_2, 1, 1) & \leftarrow choose(fink_1), choose(fink_2)
\end{array}$$

[1] For some applications it may be useful to distinguish these two alternatives. This is left to future work.

[2] Note that. motivated by the examples we want to handle, the treatment of irrelevance here differs from that in [1].

[3] The numerical condition alone does not guarantee transitivity, therefore this somewhat more involved definition is needed.

The preference program, P_{pref}, of P consists of the rules:

$$\begin{array}{l} r_1(A_2) : max\{U_1 : u(A_1, A_2, U_1, U_2)\} \leftarrow action_2(A_2) \\ r_2(A_1) : max\{U_2 : u(A_1, A_2, U_1, U_2)\} \leftarrow action_1(A_1) \end{array}$$

These rules have 4 relevant instances, namely $r_1(quiet_2)$, $r_1(fink_2)$, $r_2(quiet_1)$ and $r_2(fink_1)$. P_{gen} has four answer sets which are:

$$\begin{array}{l} I_1 = \{\ choose(quiet_1),\ choose(quiet_2),\ u(quiet_1, quiet_2, 2, 2)\ \} \\ I_2 = \{\ choose(quiet_1),\ choose(fink_2),\ u(quiet_1, fink_2, 0, 3)\ \} \\ I_3 = \{\ choose(fink_1),\ choose(quiet_2),\ u(fink_1, quiet_2, 3, 0)\ \} \\ I_4 = \{\ choose(fink_1),\ choose(fink_2),\ u(fink_1, fink_2, 1, 1)\ \} \end{array}$$

It can be easily verified that

$$\begin{array}{llll} I_1 \models_{irr} r_1(quiet_2), & I_1 \models_{irr} r_1(fink_2), & I_1 \models_{irr} r_2(quiet_1), & I_1 \models_{irr} r_2(fink_1). \\ I_2 \models_{irr} r_1(quiet_2), & I_2 \models_{irr} r_1(fink_2), & I_2 \models_1 r_2(quiet_1), & I_2 \models_{irr} r_2(fink_1). \\ I_3 \models_1 r_1(quiet_2), & I_3 \models_{irr} r_1(fink_2), & I_3 \models_{irr} r_2(quiet_1), & I_3 \models_{irr} r_2(fink_1). \\ I_4 \models_{irr} r_1(quiet_2), & I_4 \models_1 r_1(fink_2), & I_4 \models_{irr} r_2(quiet_1), & I_4 \models_1 r_2(fink_1). \end{array}$$

Thus, I_1 is the least preferred answer set and I_2, I_3 are both preferred over I_1. However, I_4 is the maximally cardinality preferred answer set and encodes the only Nash equilibrium of the game. Note that contrary to the representation of equilibria in [5], P_{pref} works for arbitrary 2-person games, not just for the prisoner dilemma, which is much more in the spirit of ASP. Moreover, a generalization to n-person games with $n > 2$ is straightforward.

References

1. Brewka, G., Niemelä, I., Truszczynski, M.: Answer set optimization. In: Proc. IJCAI, pp. 867–872 (2003)
2. Brewka, G.: Logic programming with ordered disjunction. In: Proc. AAAI, pp. 100-105 (2002)
3. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence 175(1), 278–298 (2010)
4. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
5. Foo, N., Meyer, T., Brewka, G.: LPOD answer sets and nash equilibria. In: Maher, M.J. (ed.) ASIAN 2004. LNCS, vol. 3321, pp. 343–351. Springer, Heidelberg (2004)
6. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. TPLP 7, 355–375 (2007)
7. Sakama, C., Inoue, K.: Prioritized logic programming and its application to common-sense reasoning. Artificial Intelligence 123(1-2), 185–222 (2000)
8. Schaub, T., Wang, K.: A comparative study of logic programming with preference. In: Proc. IJCAI, pp. 597-602 (2001)

Optimizing the Distributed Evaluation of Stratified Programs via Structural Analysis*

Rosamaria Barilaro, Francesco Ricca, and Giorgio Terracina

Department of Mathematics, University of Calabria, Italy
{barilaro,ricca,terracina}@mat.unical.it

Abstract. The attention received by query optimization is constantly growing, but efficiently reasoning over natively distributed data is still an open issue. Three main problems must be faced in this context: *(i)* rules to be processed may contain many atoms and may involve complex joins among them; *(ii)* the original distribution of input data is a fact and must be considered in the optimization process; *(iii)* the integration of the reasoning engine with DBMSs must be tight enough to allow efficient interactions but general enough to avoid limitations in kind and location of databases. This paper provides an optimization strategy based on structural analysis facing these issues.

1 Introduction

The basic problem of querying distributed deductive databases has been studied in the literature [12]; also distributing the evaluation of Answer Set Programs received some attention [2]. The proposed techniques for program evaluation mostly focus on shipping the data for distributing the evaluation of rules (e.g., according to the *copy-and-constraint* [12] technique), and heuristically balancing the load on available machines [5]; however, these approaches usually do not consider as founding hypothesis that data is natively distributed. Moreover, an important role for efficiently evaluating a rule is also played by its structure; in particular, the interactions among join variables might (or might not) affect evaluation costs [11]. This follows from known results regarding conjunctive query evaluation, by considering that the evaluation of a rule body is similar to the evaluation of a conjunctive query. Structural methods [4,9] allow the transformation of a conjunctive query into a (tree-like) set of sub queries, allowing efficient evaluations, e.g., by the well-known Yannakakis algorithm [13]. Despite the attention received by structural query optimization in the field of databases, the specialization of such techniques for querying natively distributed data, has not been considered much. Moreover, to the best of our knowledge, their application for distributed evaluation of logic programs has not been investigated.

In this paper we focus on a scenario, where data *natively* resides on different *autonomous* sources and it is necessary to deal with reasoning tasks via logic programming. Three main problems must be faced in this context: *(i)* rules to be processed may

* This work has been partially supported by the Calabrian Region under PIA (Pacchetti Integrati di Agevolazione industria, artigianato e servizi) project DLVSYSTEM approved in BURC n. 20 parte III del 15/05/2009 - DR n. 7373 del 06/05/2009.

contain many atoms and may involve complex joins among them; *(ii)* the original distribution of input data is a fact and this must be considered in the optimization process; *(iii)* the integration of the reasoning engine with DBMSs must be tight enough to allow efficient interactions but general enough to avoid limitations in kind and location of databases. This paper aims to be a first step toward this direction. Specifically, issue *(i)* is addressed by an optimization strategy based on structural analysis, and in particular on a hypertree decomposition method [9] that we extend in order to take into account both the distribution of the sources and possible negation in rule bodies. The proposed technique includes strategies for deciding whether to ship rules or data among distributed sites; this addresses issue *(ii)*. Finally, we adopt DLV^{DB} [10] as core reasoning engine, which allows to transparently evaluate logic programs directly on commercial DBMSs; this answers issue *(iii)*.

In this paper we concentrate on the evaluation of normal stratified logic programs [1,8] and we assume the reader familiar with logic programming.

In order to asses the effectiveness of the proposed approach, we carried out a preliminary experimental activity, on both real world and synthetic benchmarks, for comparing the performance of our approach with commercial solutions. Obtained results, reported in the paper, are encouraging and confirm our intuitions.

We next present our optimization approach for programs composed of one single rule first, and then we generalize to generic programs.

2 Single Rule Optimized Evaluation

A single logic rule r can be seen as a conjunctive query (possibly with negation), whose result must be stored in the head predicate h.

The optimized evaluation of r starts from the computation of its structural decomposition, based on an extension of the algorithm `cost-k-decomp`, introduced in [9]; then the output of the algorithm is a hypertree which is interpreted as a distributed query plan. In more detail, `cost-k-decomp` has been extended as follows.

In order to take into account data distribution within the computation of the decomposition, each node p of the hypertree $HD = \langle N, E \rangle$ is labelled with the database where the partial data associated with it are supposed to reside. Formally, let $Site(p)$ denote the site associated with the node p in HD and let $net(Site(p), Site(p'))$ be the unitary data transfer cost from $Site(p)$ to $Site(p')$ (clearly, $net(Site(p), Site(p)) = 0$). Let $\lambda(p)$ be the set of atoms referred by p, and $\chi(p)$ the variables covered by p. $Site(p)$ is chosen among the databases where the relations in $\lambda(p)$ reside by computing: $h_m = arg\ min_{h_i \in \lambda(p)} \{\Sigma_{h_j \in \lambda(p)} |rel(h_j)| \times net(Site(h_j), Site(h_i))\}$. Then, $Site(p) = Site(h_m)$.

In order to handle the presence of negated atoms in r, the construction of valid hypertrees has been modified in such a way that each node containing a negated atom is a leaf node and it does not contain other atoms. This is needed to isolate negated atoms in order to specifically handle them in the overall computation. However, observe that since r is safe these constraints are not actual limitations for computing valid hypertrees.

In order to take into account both data transfer costs and the impact of negated predicates on the overall computational costs, the cost function adopted in `cost-k-decomp`

```
Procedure SolveRule(Hypertree Node $p$)
begin
if $p$ is a leaf and $\lambda(p)$ contains only one relation $h$ then
    if $p$ has a father $p'$ then project $h$ on $\chi(p')$
    Store the result in a relation $h_p$ in $Site(p)$ and Tag the node $p$ as solved
else if $p$ is a leaf and $\lambda(p)$ contains relations $b_1, \cdots, b_k$ then
    Set the working database of DLV$^{DB}$ as $Site(p)$
    Transfer each $b_i$ not in $Site(p)$ with the USE clause of DLV$^{DB}$
    Call DLV$^{DB}$ to evaluate the rule $h_p :- b_1, \cdots, b_k$ on $Site(p)$
    if $p$ has a father $p'$ then project $h_p$ on $\chi(p')$
    Store $h_p$ in $Site(p)$ and Tag the node $p$ as solved
else
    for each $p'_i \in \{p'_1 \cdots, p'_m\}$ being an unsolved child of $p$
        Launch a process executing SolveRule($p'_i$);
    Synchronize processes (barrier)
    Set the working database of DLV$^{DB}$ as $Site(p)$
    Let $b_1, \cdots b_k$ be the relations in $p$
    Transfer each $b_i$ and $p'_i$ not in $Site(p)$ with the USE clause of DLV$^{DB}$
    Call DLV$^{DB}$ to evaluate the rule $h_p :- b_1, \cdots, b_k, h_{p'_1} \cdots, h_{p'_l}, not\ h_{p'_{l+1}}, \cdots, not\ h_{p'_m}$
        where $p'_1, \cdots, p'_l$ are child nodes of $p$ corresponding to positive atoms in $r$
        and $p'_{l+1}, \cdots, p'_m$ are child nodes of $p$ corresponding to negated atoms in $r$
    if $p$ has a father $p'$ then project $h_p$ on $\chi(p')$
    Store $h_p$ in $Site(p)$ and Tag the node $p$ as solved
end else;
end Procedure;
```

Fig. 1. Procedure *SolveRule* for evaluating a rule optimized by hypertree decomposition

is changed to: $\omega^S_{\mathcal{H}}(HD) = \Sigma_{p \in N}(est(E(p)) + min_{h_i \in \lambda(p)}\{\Sigma_{h_j \in \lambda(p)}|rel(h_j)| \times net(Site(h_j), Site(h_i))\} \ + \Sigma_{(p,p') \in E}(est^*(p,p') \ + \ est(E(p')) \ \times \ net(Site(p'), Site(p))))$
where

$$est^*(p,p') = \begin{cases} est(E(p)) - est(E(p) \bowtie E(p')) & \text{if } p' \text{ is negated in } r \\ est(E(p) \bowtie E(p')) & \text{otherwise} \end{cases}$$

Here, if $\lambda(p)$ contains only one relation h and p is a leaf in HD, $est(E(p))$ is exactly the number of tuples in h; otherwise, it estimates the cardinality of the expression associated with p, namely $E(p) = \bowtie_{h \in \lambda(p)} \ \Pi_{\chi(p)} rel(h)$. Let R and S be two relations, $est(R \bowtie S)$ is computed as:

$$est(R \bowtie S) = \frac{est(R) \times est(S)}{\Pi_{A \in attr(R) \cap attr(S)} max\{V(A,R), V(A,S)\}}$$

where $V(A,R)$ is the selectivity of attribute A in R. For joins with more relations one can repeatedly apply this formula to pair of relations according to a given evaluation order. A more detailed discussion on this estimation can be found in [11].

We are now able to describe how the evaluation of a rule r is carried out in our approach: *(i)* Create the hypergraph $\mathcal{H}_r$ for r. *(ii)* Call `cost-k-decomp` extended as described above on $\mathcal{H}_r$, and using $\omega^S_{\mathcal{H}}(HD)$. *(iii)* Tag each node of the obtained hypertree HD_r as *unsolved*. *(iv)* Call the Procedure *SolveRule* shown in Figure 1 to compose from HD_r a distributed plan for r and execute it.

Intuitively, once the hypertree decomposition is obtained, *SolveRule* evaluates joins bottom-up, from the leaves to the root, suitably transferring data if the sites of a child node and its father are different. Independent sub-trees are executed in parallel processes. In this way, the evaluation can benefit from parallelization.

It is worth pointing out that the benefits of parallelization possibly exploited in *SolveRule* are currently not considered in the overall cost function $\omega_{\mathcal{H}}^{S}(HD)$; in fact, the choices that can maximize parallelization are orthogonal to those aiming at minimizing join and data transfer costs. As a consequence, in a first attempt, we decided to privilege the optimization of join costs, while taking benefit of possible parallelization. Observe that this choice allows our approach to be safely exploited also when all relations reside on the same database.

3 Evaluation of the Program

Three main kinds of optimization characterize our approach, namely *(i)* rule unfolding optimization, *(ii)* inter-components optimization, *(iii)* intra-component optimization.

In many situations, when evaluating a program, one is interested in the results of only a subset of the predicates in the program. As far as stratified programs are concerned, the specification of a filter basically corresponds to specifying a relevant subportion of the program. Query oriented optimizations can be exploited in this case. Since the rule optimization strategy presented in the previous section is particularly suited for rules having long bodies, in presence of filters (or queries) in the program we adopt a standard unfolding optimization [8] step that has the effect of making rule bodies as long as possible and, possibly, reduce their number. Program unfolding proceeds as usual [8], it starts from the predicates specified in the filter and, following the dependencies, properly substitutes the occurrence of atoms in the body by their definitions.

The *Inter-Components optimization* [3], consists of dividing the input (possibly unfolded) program $\mathcal{P}$ into subprograms, according to the dependencies among the predicates occurring in it, and by identifying which of them can be evaluated in parallel. Indeed, if two components A and B, do not depend on each other, then the evaluation of the corresponding program modules can be performed simultaneously, because the evaluation of A does not require the data produced by B and vice versa.

The *Intra-Component optimization* [3], allows for concurrently evaluating rules involved by the same component. Observe that rules in $\mathcal{P}$ may be recursive. A rule r occurring in a *module* of a component C (i.e., defining some predicate in C) is said to be *recursive* if there is a predicate $p \in C$ occurring in the positive body of r; otherwise, r is said to be an *exit rule*. Recursive rules are evaluated following a semi-naïve schema [11]. Specifically, for the evaluation of a module M, first all exit rules are processed in parallel, afterward, recursive rules are processed several times by applying a semi-naïve evaluation technique in which, at each iteration n, the instantiation of all the recursive rules is performed concurrently (synchronization occurs at the end of each iteration). Both exit and recursive rules are evaluated with the optimization schema described in Section 2.

4 Experiments

In this section we present preliminary results of the experiments we carried out by exploiting a prototypical implementation of our approach. In the following, after describing compared methods and benchmark settings, we address tests on both a real world scenario and synthetic benchmarks from OpenRuleBench [7].

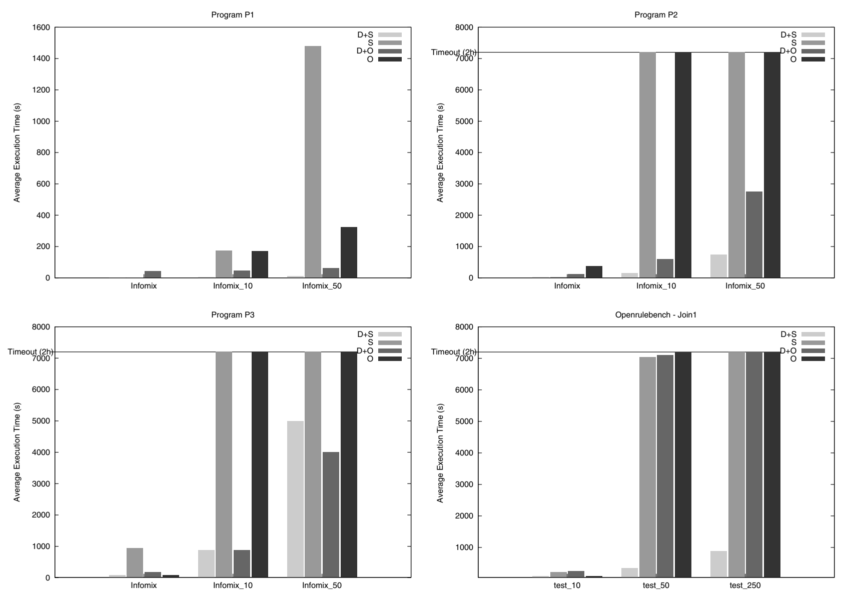


Fig. 2. Tests results

Compared Methods and Benchmark Settings. We compared our approach with two well known DBMSs allowing to manipulate and query distributed data, namely Oracle and SQLServer. Since we are interested in comparing the behaviour of our approach with commercial DBMSs, we evaluated the programs with our approach and the corresponding (set of) SQL queries with the DBMSs. SQLServer allows to query distributed data via *linked servers*, whereas Oracle provides *database links*. In our approach DLV^{DB} has been coupled with both SQLServer and Oracle. The hardwares used for the experiments are rack mount HP ProLiant DL120 G6 equipped with Intel Xeon X3430, 2.4 GHz, with 4 Gb Ram, running Windows 2003 Server Operating System. We set a time limit of 120 minutes after which the execution of a system has been stopped. For each benchmark we have averaged the results of three consecutive runs after the first (which was not considered in order to factor out internal DBMSs optimizations like caching). In the graphs (see Figure 2), we report the execution times required by DLV^{DB} coupled with SQLServer (D+S), SQLServer (S), DLV^{DB} coupled with Oracle (D+O), and Oracle (O).

Tests on a real world scenario. We exploited the real-world data integration framework developed in the INFOMIX project (IST-2001-33570) [6], which integrates data from a real university context. In particular, considered data sources were made available by the University of Rome "La Sapienza". We call this data set **Infomix** in the following. Moreover, we considered two further data sets, namely **Infomix-x-10** and **Infomix-x-50** storing 10 and 50 copies of the original database, respectively. It holds that **Infomix** $\subset$ **Infomix-x-10** $\subset$ **Infomix-x-50**. We then distributed the Infomix data sets over 5 sites and we compared the execution times of our prototype with the behavior of the two DBMSs on three programs. Note that, the rules composing the above three programs were unfolded w.r.t. output predicates and the corresponding rules rewritten as SQL queries to be tested on both Oracle and SQLServer.

The results of our experiments are presented in Figure 2. From the analysis of this figure it is possible to observe that our approach allows to obtain significant scalability improvements. In fact, while for the smallest data set times vary from few seconds (**P1**) to hundreds of seconds (**P2** and **P3**), DBMSs exceed the timeout in both **P2** and **P3** already for **Infomix-x-10**. Moreover, when DBMSs do not exceed the timeout, `D+S` allows to obtain a gain up to 99% w.r.t. `S` and `D+O` up to 80% w.r.t. `O`.

Tests from OpenRuleBench. In [7] a benchmark for testing rule based systems has been presented. In this paper, we consider the program and the instances called **join1** in [7], distributing the sources in three sites. Results are shown in Figure 2; in this case, the scalability of `D+S` is impressive w.r.t. the other systems, whereas it has been quite surprising the behaviour of `D+O`. We further investigated on this and found that: *(i)* the time required for data transfers between database links in Oracle is double w.r.t. that required by linked servers in SQLServer; *(ii)* while `D+O` required almost 2 hours for **test_50**, `O` did not finish this test in 5 hours; *(iii)* we also tried to run both `D+O` and `O` on a single machine but we had to stop them after 3 hours. Thus, we had to conclude that this test was particularly tough for Oracle, independently of our optimization.

References

1. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. In: Minker [8], pp. 89–148
2. Balduccini, M., Pontelli, E., Elkhatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. Parallel Computing 31(6), 608–647 (2005)
3. Calimeri, F., Perri, S., Ricca, F.: Experimenting with Parallelism for the Instantiation of ASP Programs. Journal of Algorithms in Cognition, Informatics and Logics 63(1-3), 34–54 (2008)
4. Chekuri, C., Rajaraman, A.: Conjunctive query containment revisited, pp. 56–70 (1998)
5. Dewan, H.M., Stolfo, S.J., Hernández, M., Hwang, J.-J.: Predictive dynamic load balancing of parallel and distributed rule and query processing. In: Proc. of ACM SIGMOD 1994, pp. 277–288. ACM, New York (1994)
6. Leone, N., et al.: The INFOMIX system for advanced integration of incomplete and inconsistent data. In: Proc. of SIGMOD 2005, pp. 915–917. ACM, New York (2005)
7. Liang, S., Fodor, P., Wan, H., Kifer, M.: Openrulebench: an analysis of the performance of rule engines. In: Proc. of WWW 2009, pp. 601–610 (2009)
8. Minker, J. (ed.): Foundations of Deductive Databases and Logic Programming. Washington DC (1988)
9. Scarcello, F., Greco, G., Leone, N.: Weighted hypertree decompositions and optimal query plans. JCSS 73(3), 475–506 (2007)
10. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP 8(2), 129–165 (2008)
11. Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press, Rockvillie (1989)
12. Wolfson, O., Ozeri, A.: A new paradigm for parallel and distributed rule-processing. In: SIGMOD Conference 1990, New York, USA, pp. 133–142 (1990)
13. Yannakakis, M.: Algorithms for acyclic database schemes. In: Proc. of VLDB 1981, Cannes, France, pp. 82–94 (1981)

Contingency-Based Equilibrium Logic

Luis Fariñas del Cerro and Andreas Herzig

IRIT-CNRS, Université de Toulouse
{farinas,herzig}@irit.fr

Abstract. We investigate an alternative language for equilibrium logic that is based on the concept of positive and negative contingency. Beyond these two concepts our language has the modal operators of necessity and impossibility and the Boolean operators of conjunction and disjunction. Neither negation nor implication are available. Our language is just as expressive as the standard language of equilibrium logic (that is based on conjunction and intuitionistic implication).

1 Introduction

Traditionally, modal logics are presented as extensions of classical propositional logic by modal operators of necessity L and possibility M (often written $\Box$ and $\Diamond$). These operators are interpreted in Kripke models: triples $M = \langle W, R, V \rangle$ where W is a nonempty set of possible worlds, $R : W \longrightarrow 2^W$ associates to every $w \in W$ the set of worlds $R(w) \subseteq W$ that are accessible from w, and $V : W \longrightarrow 2^{\mathbb{P}}$ associates to every $w \in W$ the subset of the set of propositional variables $\mathbb{P}$ that is true at w. The truth conditions are:

$$\begin{aligned} M, w \Vdash \mathsf{L}\varphi &\text{ iff } M, v \Vdash \varphi \text{ for every } v \in R(w) \\ M, w \Vdash \mathsf{M}\varphi &\text{ iff } M, v \Vdash \varphi \text{ for some } v \in R(w) \end{aligned}$$

Let $\mathcal{L}_{\mathsf{L},\mathsf{M}}$ be the language built from L, M, and the Boolean operators $\neg$, $\vee$ and $\wedge$. Other languages to talk about Kripke models exist. One may e.g. formulate things in terms of strict implication $\varphi > \psi$, which has the same interpretation as $\mathsf{L}(\varphi \to \psi)$ [6]. In this paper we study yet another set of primitives that is based on the notion of *contingency*. Contingency is the opposite of what might be called 'being settled', i.e. being either necessary or impossible. Contingency of φ can be expressed in $\mathcal{L}_{\mathsf{L},\mathsf{M}}$ by the formula $\neg\mathsf{L}\varphi \wedge \neg\mathsf{L}\neg\varphi$. One may distinguish contingent truth $\varphi \wedge \neg\mathsf{L}\varphi \wedge \neg\mathsf{L}\neg\varphi$ from contingent falsehood $\neg\varphi \wedge \neg\mathsf{L}\varphi \wedge \neg\mathsf{L}\neg\varphi$. If the modal logic is at least **KT** (relation R is reflexive, characterised by the axiom $\mathsf{L}\varphi \to \varphi$) then contingent truth of φ reduces to $\varphi \wedge \neg\mathsf{L}\varphi$, and contingent falsehood of φ reduces to $\neg\varphi \wedge \neg\mathsf{L}\neg\varphi$. We adopt the latter two as our official definitions of contingency: $\mathsf{C}^+\varphi$ denotes contingent truth of φ, and $\mathsf{C}^-\varphi$ denotes contingent falsity of φ. We take these two operators as primitive, together with necessity $\mathsf{L}^+\varphi$ and impossibility $\mathsf{L}^-\varphi$. In terms of $\mathcal{L}_{\mathsf{L},\mathsf{M}}$, $\mathsf{L}^+\varphi$ is $\mathsf{L}\varphi$, $\mathsf{L}^-\varphi$ is $\mathsf{L}\neg\varphi$, $\mathsf{C}^+\varphi$ is $\varphi \wedge \neg\mathsf{L}\varphi$, and $\mathsf{C}^-\varphi$ is $\neg\varphi \wedge \neg\mathsf{L}\neg\varphi$. In our language the negation operator is superfluous because $\neg\varphi$ is going to have the same interpretation as $\mathsf{L}^-\varphi \vee \mathsf{C}^-\varphi$.

In this paper we focus on the fragment of formulas whose modal depth is at most one. The paper is organized as follows. We first give syntax and semantics and study some properties. We then show that in models with at most two points, every formula is equivalent to a formula of depth at most one. We finally establish the link with the intermediate logic of here and there as studied in answer set programming.

2 The Logic of Contingency

Our language $\mathcal{L}_{pos}$ is *without negation*, and has four primitive operators of contingent truth C^+, contingent falsehood C^-, necessity L^+ and impossibility L^-. Its BNF is:

$$\varphi ::= p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{C}^+\varphi \mid \mathsf{C}^-\varphi \mid \mathsf{L}^+\varphi \mid \mathsf{L}^-\varphi$$

where p ranges over the set of propositional variables $\mathbb{P}$. L^+ and C^+ are the positive operators and L^- and C^- are the negative operators. $\mathcal{L}^1_{pos}$ is the set of formulas of $\mathcal{L}_{pos}$ of modal depth at most 1.

The *modal depth* of a formula φ is the maximum number of nested modal operators in φ. The set of propositional variables occurring in φ is written $\mathbb{P}_\varphi$.

Given a Kripke model $M = \langle W, R, V\rangle$, the truth conditions are as follows:

$$\begin{aligned}
&M, w \Vdash \mathsf{L}^+\varphi \text{ iff } M, v \Vdash \varphi \text{ for every } v \in R(w)\\
&M, w \Vdash \mathsf{L}^-\varphi \text{ iff } M, v \nVdash \varphi \text{ for every } v \in R(w)\\
&M, w \Vdash \mathsf{C}^+\varphi \text{ iff } M, w \Vdash \varphi \text{ and } M, w \nVdash \mathsf{L}^+\varphi\\
&M, w \Vdash \mathsf{C}^-\varphi \text{ iff } M, w \nVdash \varphi \text{ and } M, w \nVdash \mathsf{L}^-\varphi
\end{aligned}$$

Validity and satisfiability are defined as usual.

The modal operators C^+ and C^- are neither normal boxes nor normal diamonds in Chellas's sense [1]. However, the following distribution properties hold.

Proposition 1. *The following equivalences are valid in the class of all models.*

$$\begin{aligned}
\mathsf{L}^+(\varphi \wedge \psi) &\leftrightarrow \mathsf{L}^+\varphi \wedge \mathsf{L}^+\psi\\
\mathsf{L}^-(\varphi \vee \psi) &\leftrightarrow \mathsf{L}^-\psi \wedge \mathsf{L}^-\psi\\
\mathsf{C}^+(\varphi \wedge \psi) &\leftrightarrow \varphi \wedge \psi \wedge (\mathsf{C}^+\varphi \vee \mathsf{C}^+\psi)\\
\mathsf{C}^-(\varphi \vee \psi) &\leftrightarrow \neg\varphi \wedge \neg\psi \wedge (\mathsf{C}^-\varphi \vee \mathsf{C}^-\psi)
\end{aligned}$$

There are no similar equivalences for the other combinations of contingency operators and Boolean connectors.

Our results in this paper are mainly for models where the accessibility relation is *reflexive*, i.e. where $w \in R(w)$ for every world w. If M is reflexive then $M, w \nVdash \varphi$ iff $M, w \Vdash \mathsf{L}^-\varphi \vee \mathsf{C}^-\varphi$ for every w in M. We can therefore define $\neg\varphi$ to be an abbreviation of $\mathsf{L}^-\varphi \vee \mathsf{C}^-\varphi$. The operators $\bot$, $\rightarrow$, and $\leftrightarrow$ can then also be defined as abbreviations: $\bot$ is $\mathsf{L}^+p \wedge \mathsf{L}^-p$, for some $p \in \mathbb{P}$; $\varphi \rightarrow \psi$ is $\mathsf{L}^-\varphi \vee \mathsf{C}^-\varphi \vee \psi$; and $\varphi \rightarrow \psi$ is $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. In reflexive models we can also define $\mathsf{M}\varphi$ as an abbreviation of $\mathsf{L}^+\varphi \vee \mathsf{C}^+\varphi \vee \mathsf{C}^-\varphi$. Moreover, our four modal operators are exclusive and exhaustive.

Proposition 2. *The formula* $\mathsf{L}^+\varphi \vee \mathsf{L}^-\varphi \vee \mathsf{C}^+\varphi \vee \mathsf{C}^-\varphi$ *and the formulas*

$$\begin{array}{lll}
\neg(\mathsf{L}^+\varphi \wedge \mathsf{L}^-\varphi) & \neg(\mathsf{L}^+\varphi \wedge \mathsf{C}^-\varphi) & \neg(\mathsf{L}^-\varphi \wedge \mathsf{C}^-\varphi)\\
\neg(\mathsf{L}^+\varphi \wedge \mathsf{C}^+\varphi) & \neg(\mathsf{L}^-\varphi \wedge \mathsf{C}^+\varphi) & \neg(\mathsf{C}^+\varphi \wedge \mathsf{C}^-\varphi)
\end{array}$$

are valid in the class of reflexive Kripke models.

Finally, note that the equivalence $\varphi \leftrightarrow \mathsf{L}^+\varphi \vee \mathsf{C}^+\varphi$ is valid in the class of reflexive models. It follows that every $\mathcal{L}_{pos}$ formula can be rewritten to a formula such that every propositional variable is in the scope of at least one modal operator.

3 Models with at Most Two Points

In the rest of the paper we consider reflexive models with at most two points. For that class we are going to establish a strong normal form.

Proposition 3. *The equivalences*

$$\begin{array}{rclcrcl}
\mathsf{L}^+\mathsf{L}^+\varphi & \leftrightarrow & \mathsf{L}^+\varphi & \qquad & \mathsf{C}^+\mathsf{L}^+\varphi & \leftrightarrow & \bot \\
\mathsf{L}^+\mathsf{L}^-\varphi & \leftrightarrow & \mathsf{L}^-\varphi & & \mathsf{C}^+\mathsf{L}^-\varphi & \leftrightarrow & \bot \\
\mathsf{L}^+\mathsf{C}^+\varphi & \leftrightarrow & \bot & & \mathsf{C}^+\mathsf{C}^+\varphi & \leftrightarrow & \mathsf{C}^+\varphi \\
\mathsf{L}^+\mathsf{C}^-\varphi & \leftrightarrow & \bot & & \mathsf{C}^+\mathsf{C}^-\varphi & \leftrightarrow & \mathsf{C}^-\varphi \\
\mathsf{L}^-\mathsf{L}^+\varphi & \leftrightarrow & \mathsf{L}^-\varphi \vee \mathsf{C}^+\varphi & & \mathsf{C}^-\mathsf{L}^+\varphi & \leftrightarrow & \mathsf{C}^-\varphi \\
\mathsf{L}^-\mathsf{L}^-\varphi & \leftrightarrow & \mathsf{L}^+\varphi \vee \mathsf{C}^-\varphi & & \mathsf{C}^-\mathsf{L}^-\varphi & \leftrightarrow & \mathsf{C}^+\varphi \\
\mathsf{L}^-\mathsf{C}^+\varphi & \leftrightarrow & \mathsf{L}^+\varphi \vee \mathsf{L}^-\varphi \vee \mathsf{C}^-\varphi & & \mathsf{C}^-\mathsf{C}^+\varphi & \leftrightarrow & \bot \\
\mathsf{L}^-\mathsf{C}^-\varphi & \leftrightarrow & \mathsf{L}^+\varphi \vee \mathsf{L}^-\varphi \vee \mathsf{C}^+\varphi & & \mathsf{C}^-\mathsf{C}^-\varphi & \leftrightarrow & \bot
\end{array}$$

are valid in the class of Kripke models having at most two points.

Proposition 3 allows to reduce every modality to a Boolean combination of modalities of length at most one (starting from outermost operators). Beyond the reduction of modalities, reflexive models with at most two points also allow for the distribution of modal operators over conjunctions and disjunctions. Proposition 1 allows us to distribute the positive operators L^+ and C^+ over conjunctions and the negative operators L^- and C^- over disjunctions. The next proposition deals with the remaining cases.

Proposition 4. *The equivalences*

$$\begin{array}{rcl}
\mathsf{L}^+(\varphi \vee \psi) & \leftrightarrow & \mathsf{L}^+\varphi \vee \mathsf{L}^+\psi \vee (\mathsf{C}^+\varphi \wedge \mathsf{C}^-\psi) \vee (\mathsf{C}^-\varphi \wedge \mathsf{C}^+\psi) \\
\mathsf{L}^-(\varphi \wedge \psi) & \leftrightarrow & \mathsf{L}^-\varphi \vee \mathsf{L}^-\psi \vee (\mathsf{C}^+\varphi \wedge \mathsf{C}^-\psi) \vee (\mathsf{C}^-\varphi \wedge \mathsf{C}^+\psi) \\
\mathsf{C}^+(\varphi \vee \psi) & \leftrightarrow & (\mathsf{C}^+\varphi \wedge \mathsf{C}^+\psi) \vee (\mathsf{C}^+\varphi \wedge \mathsf{L}^-\psi) \vee (\mathsf{L}^-\varphi \wedge \mathsf{C}^+\psi) \\
\mathsf{C}^-(\varphi \wedge \psi) & \leftrightarrow & (\mathsf{C}^-\varphi \wedge \mathsf{C}^-\psi) \vee (\mathsf{C}^-\varphi \wedge \mathsf{L}^+\psi) \vee (\mathsf{L}^+\varphi \wedge \mathsf{C}^-\psi)
\end{array}$$

are valid in the class of Kripke models having at most two points.

Distributing the modal operators over conjunctions and disjunctions results in a formula made up of modal atoms —modalities followed by a propositional variable— that are combined by conjunctions and disjunctions. These modal atoms can then be reduced by Proposition 3. If we moreover use that $\varphi \leftrightarrow \mathsf{L}^+\varphi \vee \mathsf{C}^+\varphi$ is valid in the class of reflexive models then we obtain a very simple normal form.

Definition 1. *A formula is in* strong normal form *if and only if it is built according to the following BNF:*

$$\varphi ::= \mathsf{L}^+p \mid \mathsf{L}^-p \mid \mathsf{C}^+p \mid \mathsf{C}^-p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

where p ranges over the set of propositional variables $\mathbb{P}$.

Theorem 1. *In the class of reflexive Kripke models with at most two points, every $\mathcal{L}_{pos}$ formula is equivalent to a formula in strong normal form.*

We now focus on models that are not only reflexive but also *antisymmetric*. For that class of models we give a validity-preserving translation from $\mathcal{L}_{pos}$ to propositional logic. We define two functions t^H and t^T by mutual recursion.

$$\begin{aligned}
t^H(p) &= p^H, \quad \text{for } p \in \mathbb{P} & t^T(p) &= p^T, \quad \text{for } p \in \mathbb{P}\\
t^H(\varphi \wedge \psi) &= t^H(\varphi) \wedge t^H(\psi) & t^T(\varphi \wedge \psi) &= t^T(\varphi) \wedge t^T(\psi)\\
t^H(\varphi \vee \psi) &= t^H(\varphi) \vee t^H(\psi) & t^T(\varphi \vee \psi) &= t^T(\varphi) \vee t^T(\psi)\\
t^H(\mathsf{L}^+\varphi) &= t^H(\varphi) \wedge t^T(\varphi) & t^T(\mathsf{L}^+\varphi) &= t^T(\varphi)\\
t^H(\mathsf{L}^-\varphi) &= \neg t^H(\varphi) \wedge \neg t^T(\varphi) & t^T(\mathsf{L}^-\varphi) &= \neg t^T(\varphi)\\
t^H(\mathsf{C}^+\varphi) &= t^H(\varphi) \wedge \neg t^T(\varphi) & t^T(\mathsf{C}^+\varphi) &= \bot\\
t^H(\mathsf{C}^-\varphi) &= \neg t^H(\varphi) \wedge t^T(\varphi) & t^T(\mathsf{C}^-\varphi) &= \bot
\end{aligned}$$

Theorem 2. *Let φ be a $\mathcal{L}_{pos}$ formula. φ is valid in the class of reflexive and antisymmetric models with at most two points if and only if $t^H(\varphi)$ is propositionally valid (in a vocabulary made up of the set of p^H, t^H such that p is in the vocabulary of $\mathcal{L}_{pos}$).*

If φ is in strong normal form then the translation is linear. The problem of checking validity of such formulas in reflexive and antisymmetric models is therefore in coNP.

4 Contingency and the Logic of Here-and-There

In the rest of the paper we consider models with at most two points where the accessibility relation is reflexive and *persistent*, aka *hereditary*. Just as in intuitionistic logic, we say that R is persistent if $\langle u, v\rangle \in R$ implies $V(u) \subseteq V(v)$. Such models were studied since Gödel in order to give semantics to an implication $\Rightarrow$ with strength between intuitionistic and material implication [5]. More recently these models were baptized *here-and-there models*: triples $M = \langle W, R, V\rangle$ with $W = \{H, T\}$, $V(H) \subseteq V(T)$, and $R = \{\langle H, H\rangle, \langle H, T\rangle, \langle T, T\rangle\}$[1]. Particular such models —*equilibrium models*— were investigated as a basis for answer set programming by Pearce, Cabalar, Lifschitz, Ferraris and others as a semantical framework for answer set programming [7][2].

In persistent models C^+p is unsatisfiable for every propositional variable $p \in \mathbb{P}$. In the normal form of Theorem 1 we may therefore replace every subformula C^+p by $\mathsf{L}^+p \wedge \mathsf{L}^-p$ (which is unsatisfiable in reflexive models), resulting in a formula built from modal atoms of depth one where C^+ does not occur. This observation leads also to a polynomial transformation allowing to check in reflexive and antisymmetric two-points models whether a given $\mathcal{L}_{pos}$ formula is valid in here-and-there models.

Theorem 3. *A $\mathcal{L}_{pos}$ formula φ is valid in the class of here-and-there models if and only if $\left(\bigwedge_{p\,:\,p\in\mathbb{P}_\varphi} \neg\mathsf{C}^+p\right) \rightarrow \varphi$ is valid in reflexive and antisymmetric two-points models.*

In the rest of the section we relate our language $\mathcal{L}_{pos}$ to the customary language of equilibrium logic. Let us call $\mathcal{L}^{\Rightarrow}_{pos}$ the language resulting from the addition of a binary

[1] The other reflexive and persistent models with at most two points are bisimilar to here-and-there models: (1) (pointed) two points models with a symmetric relation are bisimilar to one point models due to persistence; (2) (pointed) models where the two points are not related are bisimilar to one point models; (3) one point models are bisimilar to here-and-there models.

[2] `http://www.equilibriumlogic.net`

modal connector $\Rightarrow$ to our language $\mathcal{L}_{pos}$. The language of equilibrium logic is the fragment $\mathcal{L}^{\Rightarrow}$ of $\mathcal{L}^{\Rightarrow}_{pos}$ without our four unary modal connectors.

The truth condition for the intermediate implication $\Rightarrow$ can be written as:

$$M, w \Vdash \varphi \Rightarrow \psi \text{ iff } \forall v \in R(w),\ M, v \nVdash \varphi \text{ or } M, v \Vdash \psi$$

Given that $\mathsf{C}^+ p$ is unsatisfiable for every $p \in \mathbb{P}$, Theorem 1 shows that the above language made up of Boolean combinations of modal atoms of the form $\mathsf{L}^+ p$, $\mathsf{L}^- p$, $\mathsf{C}^- p$ is an alternative to the traditional Horn clause language.

Theorem 4. *The $\mathcal{L}^{\Rightarrow}_{pos}$ formula*

$$\varphi \Rightarrow \psi \ \leftrightarrow\ \mathsf{L}^-\varphi \vee \mathsf{L}^+\psi \vee (\mathsf{C}^-\varphi \wedge \mathsf{C}^-\psi) \vee (\mathsf{C}^+\varphi \wedge \mathsf{C}^+\psi)$$

is valid in here-and-there models.

Proof. By the truth condition of $\Rightarrow$, the formula $\varphi \Rightarrow \psi$ is equivalent to $\mathsf{L}^+(\neg\varphi \vee \psi)$. By Proposition 4 the latter is equivalent to

$$\mathsf{L}^+\neg\varphi \vee \mathsf{L}^+\psi \vee (\mathsf{C}^-\neg\varphi \wedge \mathsf{C}^+\psi) \vee (\mathsf{C}^+\neg\varphi \wedge \mathsf{C}^-\psi),$$

which is equivalent to the right-hand side. ∎

Together, theorems 4 and 3 say that instead of reasoning with an intermediate implication $\Rightarrow$ one might as well use our fairly simple modal logic of contingency having reflexive and antisymmetric two-points models. These models are not necessarily persistent. Once one has moved to that logic, one can put formulas in strong normal form (Theorem 1), apply Theorem 2, and work in classical propositional logic.

5 Contingency and Equilibrium Logic

When we talked about satisfiability in here-and-there models we took it for granted that this meant truth in some possible world w of some model M, and likewise for validity. The definitions of satisfiability and validity are more sophisticated in *equilibrium logic*.

Definition 2. *A here-and-there model $M = \langle\{H, T\}, R, V\rangle$ is an equilibrium model of φ if and only if*

- *$M, H \Vdash \varphi$,*
- *$V(H) = V(T)$, and*
- *there is no here-and-there model $M' = \langle\{H, T\}, R, V'\rangle$ such that $V'(T) = V(T)$, $V'(H) \subset V(H)$, and $M', H \Vdash \varphi$.*

An equilibrium model of φ is therefore isomorphic to a model of classical propositional logic, and moreover its here-valuation is minimal. This can be captured in our language.

Theorem 5. *The formula φ of $\mathcal{L}^{\Rightarrow}_{pos}$ has an equilibrium model iff there is $P \subseteq \mathbb{P}_\varphi$ s.th.*

$$\left(\mathsf{L}^+(\bigwedge P) \wedge \mathsf{L}^-(\bigvee(\mathbb{P}_\varphi \setminus P))\right) \to \varphi$$
$$\left(\mathsf{C}^-(\bigwedge P) \wedge \mathsf{L}^-(\bigvee(\mathbb{P}_\varphi \setminus P))\right) \to \neg\varphi$$

are both valid in reflexive and antisymmetric two-points models.

The premise of the first formula describes a here-and-there model bisimilar to the classical model P. The premise of the second formula describes all here-and-there models whose there-world matches the classical model and whose here-world is ‘less true’.

6 Conclusion

We have presented a modal logic of positive and negative contingency and have studied its properties in different classes of models. We have in particular investigated models with at most two points. We have established a link with equilibrium logics as studied in answer set programming. Our negation-free language in terms of contingency provides an alternative to the usual implication-based language. Our logic can also be seen as a combination of intuitionistic and classical implication, in the spirit of [8,3,2,4].

One of the perspectives is to study the first-order version of our logic. We can extend our translation from $\mathcal{L}_{pos}$ into propositional logic, to a translation from the first-order extension of $\mathcal{L}_{pos}$ into predicate logic as follows:

$$t^H(\forall x\varphi) = \forall x t^H(\varphi) \qquad t^T(\forall x\varphi) = \forall x t^T(\varphi)$$

This works for the first-order version of equilibrium logic with uniform domains. By means of such a translation the link with answer sets for programs with variables can be studied[3].

References

1. Chellas, B.: Modal logic: An introduction. Cambridge University Press, Cambridge (1980)
2. Došen, K.: Models for stronger normal intuitionistic modal logics. Studia Logica 44, 39–70 (1985)
3. Fariñas del Cerro, L., Raggio, A.: Some results in intuitionistic modal logic. Logique et Analyse 26(102), 219–224 (1983)
4. Fariñas del Cerro, L., Herzig, A.: Combining classical and intuitionistic logic, or: intuitionistic implication as a conditional. In: Baader, F., Schulz, K.U. (eds.) Frontiers in Combining Systems. Applied Logic Series, vol. 3, pp. 93–102. Kluwer Academic Publishers, Dordrecht (1996)
5. Heyting, A.: Die formalen Regeln der intuitionistischen Logik. Sitzungsber. Preuss. Akad. Wiss. 42-71, 158–169 (1930)
6. Hughes, G.E., Cresswell, M.J.: An introduction to modal logic. Methuen&Co. Ltd., London (1968)
7. Pearce, D.: A new logical characterisation of stable models and answer sets. In: Dix, J., Przymusinski, T.C., Moniz Pereira, L. (eds.) NMELP 1996. LNCS, vol. 1216, pp. 57–70. Springer, Heidelberg (1996)
8. Vakarelov, D.: Notes on constructive logic with strong negation. Studia Logica 36, 110–125 (1977)

[3] Thanks are due to David Pearce and the three reviewers of LPNMR-11.

Weight Constraints with Preferences in ASP*

Stefania Costantini[1] and Andrea Formisano[2]

[1] Università di L'Aquila, Italy
stefania.costantini@univaq.it
[2] Università di Perugia, Italy
formis@dmi.unipg.it

Abstract. Weight and cardinality constraints constitute a very useful programming construct widely adopted in Answer Set Programming (ASP). In recent work we have proposed RASP, an extension to plain ASP where complex forms of preferences can be flexibly expressed. In this paper, we illustrate an application of these preferences within weight/cardinality constraints. We emphasize, mainly by simple examples, the usefulness of the proposed extension. We also show how the semantics for ASP with weight constraints can be suitably extended so as to encompass RASP-like preferences, without affecting complexity.

1 Introduction

In contrast to expert knowledge, which is usually explicit, most commonsense knowledge is implicit and one of the issues in knowledge representation is making this knowledge explicit. The capability of expressing and using preferences in a formal system constitutes a significant step in this direction. It simulates a skill that every person takes for granted, being preference deeply related to a subject's personal view of the world, and driving the actions that one takes in it. From the point of view of knowledge representation, many problems are more naturally represented by flexible rather than by hard descriptions. Practically, many problems would not even be solvable if one would stick firmly on all requirements. Not surprisingly, several formalisms and approaches to deal with preferences and uncertainty have been proposed in Artificial Intelligence (such as CP-nets and preference logics, for instance) and in the specific context of computational logic [4]. Notably, many of these approaches to preference reasoning have been developed in the context of ASP [5]. ASP has the peculiarity that an ASP program may have none, one or several answer sets. These answer sets can be interpreted in various possible ways. If the program formalizes a search problem, e.g., a colorability problem, then the answer sets represents the possible solutions to the problem, namely, the possible colorings for a given graph. In knowledge representation, an ASP program may represent a formal definition of the known features of a situation/world of interest. In this case, the answer sets represent the possible consistent states of this world, that can be several whenever the formalization involves some kind of uncertainty. Also, an ASP program can be seen as the formalization of the knowledge and beliefs of a rational

* This research is partially supported by PRIN-08 and GNCS-11. Material from this manuscript composes a poster presentation at Commonsense-2011 [2].

agent about a situation, and the answer sets represent the possible belief states of such an agent, that can be several if either uncertainty or alternative choices are involved in the description [1]. Such an agent can exploit an ASP module for several purposes, such as answering questions, building plans, explaining observations, making choices, etc.

In recent work [3] we have proposed RASP, an extension to ASP where complex forms of preferences can be flexibly expressed. In that work we have considered plain ASP. In this paper, we introduce preferences into ASP extended with weight constraints [6]. A weight constraint allows one to include in the answer sets of given program an arbitrary (bounded, but variable) number of the literals indicated in the constraint itself, according to weights. Weight constraints have proved to be a very useful programming tool in many applications such as planning and configuration, and they are nowadays adopted by most ASP inference engines.

In this paper, we propose to enrich weight constraints by means of RASP-like preferences. For lack of space we explicitly consider the particular case of cardinality constraints, which are however very widely used, considering that the extension to general weight constraints is easily feasible. The advantage of introducing RASP-like preferences rather than considering one of the competing approaches is their *locality*. In fact, in RASP one may express preferences which are local to a single rule or even to a single literal. Contrasting preferences can be freely expressed in different contexts, as in our view preferences may vary with changing circumstances. Instead, most of the various forms of preferences that have been introduced in ASP [4, 7] are based on establishing priorities/preferences among rules or preferences among atoms which are anyway globally valid in given program. A weight constraint represents a local context where RASP-like preferences find a natural application.

2 RASP-Like Preferences in Cardinality Constraints

Let us recall some basic notion about weight/cardinality constraints and the extension RASP. The reader is referred to [6, 3] for much detailed treatments. A *Weight Constraint* is of the form:

$$l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \textit{not}\ a_{n+1} = w_{a_{n+1}}, \ldots, \textit{not}\ a_{n+m} = w_{a_{n+m}}\} \leq u$$

where the a_is are atoms. Each literal in a constraint has an associated numerical weight. The numbers l and u are the lower and upper bounds of the constraint. A weight constraint is satisfied by a set of atoms S if the sum of weights of those literals occurring in the constraint that are satisfied by S is in $[l, u]$. A *Cardinality Constraints* is a weight constraints such that all weights are equal to one. A shorthand form is provided:

$$l\, \{a_1, \ldots, a_n, \textit{not}\ a_{n+1}, \ldots, \textit{not}\ a_{n+m}\}\, u$$

A rule has the form $C_0 \leftarrow C_1, \ldots, C_n.$, where the C_is are weight/cardinality constraints and a program is a set of such rules.

Deciding whether a ground program has an answer set is NP-complete, and computing an answer set is FNP-complete [6].

To compactly specify sets of literals in a constraint, one can exploit variables and *conditional literals*. Let us illustrate this notation by means of an example. Assume that you wish to state that a meal is composed of at least two and at most three courses. This may be expressed by the following cardinality constraint.

$$2\{in\_menu(X,C) : course(C)\}3 \leftarrow meal(X).$$

Hence, provided this knowledge base:

$$\begin{array}{llll} meal(lunch). & coeliac. & course(cake). & course(pasta) \leftarrow not\ coeliac. \\ meal(dinner). & & course(fruit). & course(meat). \end{array}$$

the ground version of the program becomes as follows (simplified because of the truth of $meal(lunch)$ and $meal(dinner)$):

$$2\{in\_menu(lunch, meat), in\_menu(lunch, cake), in\_menu(lunch, fruit)\}3.$$
$$2\{in\_menu(dinner, meat), in\_menu(dinner, cake), in\_menu(dinner, fruit)\}3.$$

RASP is an extension to the ASP framework that allows for the specification of various kinds of non-trivial preferences. In full RASP, quantities for ingredients and products are allowed to be specified, and resources might be *consumed/produced*. However, in this paper we neglect the aspects of RASP related to resources in order to concentrate on preferences. Next definition adapts to our case the main constructs of RASP [3].

Definition 1. *Let $s_1, \dots, s_k$ be $k > 0$ either distinct constants or distinct atoms, and let $L_1, \dots, L_n$ be literals. Then,*

- *a* preference-list *(*p-list*) is of the form:* $s_1 > \cdots > s_k$
- *a* conditional p-list *(*cp-list*) is of the form:* $(s_1 > \cdots > s_k$ pref_when $L_1, \dots, L_n)$

Each of the s_i has degree of preference *i in the p-list/cp-list.*
Let $q_1, \dots, q_k$ be $k > 0$ atoms and let $pred$ be a binary program predicate. Then,

- *a* p-set *is of the form:* $\{q_1, \dots, q_k \mid pred\}$.

Intuitively, a p-list expresses a linear preference among the s_is. A cp-list specifies that such a preference should be applied only when all $L_1, \dots, L_n$ are satisfied. Otherwise, if any of the L_i does not hold, no preference is expressed. P-sets are a generalization of p-lists that allows one to use any binary relation (not necessarily a partial order) in expressing (collections of alternative) p-lists. The program predicate $pred$ is supposed to be defined elsewhere in the program where the p-set occurs.

It might be useful to enhance the modeling power of cardinality constraints by exploiting RASP-like preferences. In the rule below, we reconsider menus and courses and state, by means of a p-list, that pasta is preferred over meat.

$$2\{in\_menu(X,C) : course(C) \mid in\_menu(X, pasta) > in\_menu(X, meat)\}3 \leftarrow meal(X).$$

where constants occurring in the p-list are among the possible values of variable C, i.e., are elements of the domain of $course$. Notice that we do not need to express preferences over all possible values of C (namely, we stated we prefer to include pasta rather than meat, while we are indifferent about the third course). Preference is "soft" in the sense that pasta will be chosen if available, otherwise meat will be selected, again if available, otherwise some other course will be selected anyway. Here we extend our example by employing a cp-list to state that in summer fruit is preferred over cake:

$$2\{in\_menu(X,C) : course(C) \mid in\_menu(X, fruit) > in\_menu(X, cake)$$
$$\text{pref\_when } summer\}3 \leftarrow meal(X).$$

Finally, we may employ p-sets to state that we prefer the less caloric courses:

$$2\{in\_menu(X, C) : course(C) \mid less\_caloric[X]\}3 \leftarrow meal(X).$$

Notice that $less\_caloric$ is, according to Def. 1, a binary predicate. The notation $less\_caloric[X]$ means that the comparison is on pairs of distinct instances of variable X. This specification is necessary as different domain predicates defined over different variables may occur in constraints: this requires to indicate the variable to be considered for defining a p-set. Moreover, as Def. 2, to be seen, specifies, multiple preference are allowed in a constraint. Here, the p-set actually occurring (implicitly) in the constraint is $\{pasta, meat, fruit, cake : less\_caloric\}$.

This kind of cardinality constraints is called *p-constraint*. P-constraints may occur in the head of program rules. In general, for expressing preferences one may employ any variable occurring in the rule where the constraint appears. Note that p-lists can be defined both on constants and atoms. For instance, the next rule states that Italian food is preferred to Chinese one:

$$2\{in\_menu(X, C) : course(C) : italian(C) > chinese(C)\}3 \leftarrow meal(X).$$

The general form of non-ground p-constraints is the following

Definition 2. *A p-constraint is of the form:*

$$l\{a_1, \dots, a_n, \dots, not\ a_{n+1}, \dots, not\ a_{n+m} : D \mid C_p\}u$$

where the a_is are atoms, D is a set of atoms (concerning domain predicates), and C_p is a list of preference specifications possibly including (c)p-lists and binary predicates defining p-sets. Each two specifications in C_p are defined on distinct variables.

The purpose of the set of atoms D consists in defining the domains of the variables occurring in the a_is, as happens for the standard definition of weight constraints [6]. Then, a program rule has the form: $C_0 \leftarrow C_1, \dots, C_n,$ where C_0 is a p-constraint and the C_is are weight/cardinality constraints. A program is a set of rules.

Notice that preferences may occur only in the heads of rules, i.e., preferences might influence what should be derived from a rule.

In future work, we intend to overcome the limitation of preference specifications in C_p to be disjoint, in order to admit interacting preferences and priorities among preferences. A first step, would be to allow (c)p-lists to be defined over p-sets. For instance, referring to the above example of preferring less caloric courses, an extension would be to allow one to prefer less caloric courses in the first place, the best in quality in the second place, and the less expensive in the third place. I.e., we would allow expressions such as $less\_caloric[X] > better\_quality[X] > less\_expensive[X]$.

3 Semantics

In this section, we introduce an extension to the semantics of weight constraints as specified in [6], so as to accommodate p-constraints. We implicitly consider the ground version of given program Π. By adapting to weight constraints the approach developed for RASP in [3], we introduce a function aimed at encoding the possible orderings on domain elements, according to the preferences expressed through p-constraints. The key point of the semantic modification that we propose is exactly to exploit such a function to reorder the atoms occurring in the p-constraints of Π. Then a (candidate) answer set is accepted only if it models the most preferred atoms.

Some preliminary notions are due. Given a collection $\mathcal{S}$ of non-empty sets, a *choice function* $c(\cdot)$ for $\mathcal{S}$ is a function having $\mathcal{S}$ as domain and such that $c(s) \in s$ for each s in $\mathcal{S}$. In other words, $c(\cdot)$ chooses exactly one element from each set in $\mathcal{S}$. Given a binary predicate symbol p and a set of ground atoms I, consider all the atoms of the form $p(a,b)$ in I. Let $I_{|p}$ denote the transitive closure of the set $\{p(a,b) \mid p(a,b) \in I\}$. Namely, $I_{|p}$ is the smallest set such that for all a,b,c it holds that $\big(p(a,b) \in I \vee (p(a,c) \in I_{|p} \wedge p(c,b) \in I_{|p})\big) \rightarrow p(a,b) \in I_{|p}$.

A given answer set might satisfy one or more of the atoms occurring in a p-list (resp., cp-list, p-set). Each atom q occurring in such a p-list has a degree of preference i associated with. We introduce a function ch to represent each pair $\langle q, i\rangle$ occurring in a p-list (resp., cp-list, p-set) of a p-constraint. In particular, for p-lists we put $ch(q_1 > \cdots > q_k, I) = \big\{\{\langle q_1, 1\rangle, \ldots, \langle q_k, k\rangle\}\big\}$.

For a cp-list $q_1 > \cdots > q_k$ pref_when $L_1, \ldots, L_n$ we put:

$$ch(r,I)=\begin{cases} ch(q_1 > \cdots > q_k, \emptyset) & \text{if } I \text{ satisfies } L_1,\ldots,L_n \\ \big\{\{\langle q_1, i_1\rangle, \ldots, \langle q_k, i_k\rangle\} \mid \{i_1,\ldots,i_k\}=\{1,\ldots,k\}\big\} & \text{otherwise} \end{cases}$$

The case of a p-set $ps = \{q_1, \ldots, q_k \mid p\}$ is slightly more complicated because p represents a collection of alternative p-lists, each one potentially exploitable in a given answer set I. Let us denote such a collection of p-lists as follows:

$$\begin{aligned} PLists(ps, I) = \big\{\ & q_{i_1} > \cdots > q_{i_n} \mid \langle 1,\ldots,n\rangle \text{ is a maximal prefix of } \langle 1,\ldots,k\rangle \\ & \text{such that } \forall j,h\,(j < h \rightarrow p(q_{i_h}, q_{i_j}) \notin I_{|p})\big\} \end{aligned}$$

Then, we define $ch(ps, I) = \bigcup_{pl \in PLists(ps,I)} ch(pl, I)$. The definition of ch is then extended to rules by putting: $ch(\gamma, I) = \{ch(\ell, I) \mid \ell \text{ in the head of } \gamma\}$.

Finally, we associate to each rule γ, the set $\mathcal{R}(\gamma, I)$ of sets. Each $X \in \mathcal{R}(\gamma, I)$ is a collection of sets, each one having the form $\{\langle q_1, 1\rangle, \ldots, \langle q_k, k\rangle\}$, where q_i is an atom and i is a degree of preference. Given such an X, we say that each q such that $\langle q, 1\rangle \in x$ for some $x \in X$ is a *most preferred* element for X. Note that, each of the sets $\{\langle q_1, 1\rangle, \ldots, \langle q_k, k\rangle\}$ belonging to X encodes an ordering (i.e., a preference) on the atoms of one of the p_γ p-lists (resp., cp-lists, p-sets) occurring in γ. (Hence, for a fixed rule γ, each of the sets X in $\mathcal{R}(\gamma, I)$ has cardinality equals to p_γ.)

$$\mathcal{R}(\gamma, I) = \big\{\{c(s) \mid s \text{ in } ch(\gamma, I)\} \mid \text{for } c \text{ choice function for } ch(\gamma, I).\big\}$$

where c ranges on all possible choice functions for $ch(\gamma)$.

A candidate answer set S is actually an answer set if it chooses from each ground p-constraint C in the head of a rule γ, the most preferred elements (whatever their number is) according to some $X \in \mathcal{R}(\gamma, S)$. More formally, for a program Π, we have:

Definition 3. *A set of atoms S is an answer set for Π if it holds that:*

- *S is an answer set of Π according to Definitions 2.7 and 2.8 in [6].*
- *For every p-constraint C, head of a rule $\gamma \in \Pi$, S includes all the most preferred elements of C, w.r.t. at least one of the $X \in \mathcal{R}(\gamma, S)$.*

Notice that we do not have to consider interaction among different rules: our preferences are in fact *local* to the p-constraint where they occur. Different p-constraints can be defined over different, even contrasting, preferences.

It is easy to get convinced that function ch can be computed in polynomial time. Therefore obtain the following results, that guarantee that the further modeling power that we offer implies no computational expense.

Theorem 1. *Let Π be a ground p-program. Then,*

- *deciding whether Π has answer sets is NP-complete;*
- *deciding whether a set X of atoms is an answer set of Π is NP-complete.*

If one would now like to choose the "best preferred" answer sets, a *preference criterion* should be provided to compare answer sets. Such a criterion should impose an order on the collection of answer sets by reflecting the preference degrees in the (c)p-lists. In a sense, any criterion should aggregate/combine all "local" partial orders to obtain a global one. General techniques for combining preferences have been proposed, criteria have been also studied w.r.t. the specific framework of Logic Programming. The complexity of finding the best preferred answer sets varies according to the selected criterion (see [2, 3] and the references therein).

4 Concluding Remarks

In this paper we have presented an approach to express preferences in ASP cardinality constraints. Future work includes: the introduction of preferences among sets of options; the extension of preference treatment to the general form of weight constraints. The main point however is the full integration of weight constraints and other forms of aggregates with RASP, i.e., the introduction of resource usage in p-constraints and in their future evolutions. In fact, as mentioned, weight constraints have proved useful in many applications, among which configuration. RASP is very promising in this direction, as one may specify not only the qualitative aspects of processes, but also the quantitative aspects that are in many cases of some or great importance. Preferences are often related to resources, as an agent may prefer to consume (or, more generally, to "invest") some resources rather than others, and may also have preferences about what one should try to obtain with the available resources. Consumption or production of some resource may have a relevance, that can be expressed by the weights in weight constraint. This kind of extended formalism can find suitable applications in the realm in bio-informatics, where reactions involve quantities, weights and byproducts, and may happen according to complex preferences.

References

[1] Capotorti, A., Formisano, A.: Comparative uncertainty: theory and automation. Mathematical Structures in Computer Science 18(1) (2008)

[2] Costantini, S., Formisano, A.: Augmenting weight constraints with complex preferences. In: Proc. of Commonsense 2011. AAAI Press, Menlo Park (2011)

[3] Costantini, S., Formisano, A., Petturiti, D.: Extending and implementing RASP. Fundamenta Informaticae 105(1-2) (2011)
[4] Delgrande, J., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Computational Intelligence 20(12) (2004)
[5] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of the ICLP/SLP 1988. MIT Press, Cambridge (1988)
[6] Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2) (2002)
[7] Van Nieuwenborgh, D., Vermeir, D.: Preferred answer sets for ordered logic programs. Theory and Practice of Logic Programming 6(1-2) (2006)

Parametrized Equilibrium Logic

Ricardo Gonçalves and José Júlio Alferes*

CENTRIA, Universidade Nova de Lisboa, Portugal

Abstract. Equilibrium logic provides a logical foundation for the stable model semantics of logic programs. Recently, parametrized logic programming was introduced with the aim of presenting the syntax and natural semantics for parametrized logic programs, which are very expressive logic programs, in the sense that complex formulas are allowed to appear in the body and head of rules. Stable model semantics was defined for such parametrized logic programs. The aim of this paper is to introduce a parametrized version of equilibrium logic that extends parametrized logic programs to general theories, and to show how these can be used to characterize and to study strong equivalence of temporal logic programs.

1 Introduction

Equilibrium logic [7], and its monotonic base – the logic of Here-and-There (HT) [6] – provide a logical foundation for the stable models semantics of logic programs [4], in which several important properties of logic programs under the stable model semantics can be studied. In particular, it can be used to check strong equivalence of programs (i.e. that two programs will remain equivalent independently of whatever else we add to both of them), by checking logical equivalence in Here-and-There.

Equilibrium logic also allows for the extension of the stable model semantics to a language allowing arbitrary sets of formulas [3]. In this extension of stable models to general theories, the (disjunctive) logic programming rule symbol "$\leftarrow$" (resp. ",", ";") is equated with implication "$\Rightarrow$" (resp. "$\wedge$", "$\vee$") in equilibrium logic. Moreover, logic programming's default negation is equated with the negation $\neg$ of equilibrium logic. Of course, by doing so the classical connectives of conjunction, disjunction, implication and negation are lost in such general theories. This fact was recently noticed in [11,12] where equilibrium logic is extended for sets of formulas, called general default logic, that besides the equilibrium logic connectives also allows for having classical propositional connectives, e.g. allowing to differentiate logic programming disjunction $A;B$, which indicates that either A is known to be true or B is known to be true, from the classical disjunction $A \vee B$ which does not necessarily requires one of the propositions to be known (see [11,12] for details). This generalisation is proven in [11] to be

* The first author was supported by FCT under the postdoctoral grant SFRH/BPD/47245/2008.

more general than Reiter's default logic [10], and able to express rule constraints, generalised closed world assumption and conditional defaults.

Recently, parametrized logic programming was introduced [5], having in common with [11,12] the motivation of providing a meaning to theories combining both logic programming connectives with other logic connectives, and allowing complex formulas using these connectives to appear in the head and body of a rule. Although [5] is less general than [11,12] in the sense that it does not allow complex rule formulas to be built, it is more general in the sense that allows to deal with formulas other than classical propositional logic (CPL) formulas.

The aim of this paper is to introduce a parametrized version of equilibrium logic (Section 2) which generalizes both the approaches in [5] and [11,12], by allowing complex rule formulas along with the use of formulas other than those of CPL. As an example of the usefulness of this extra generality, we experiment (Section 3) with linear temporal logic (LTL) [9]. We compare the obtained logic with a temporal extension of Here-and-There logic, THT, introduced in [1].

2 Parametrized Equilibrium Logic

In this section we present a parametrized version of equilibrium logic and of its monotonic base, the logic of Here-and-There (HT) [6], generalising parametrized logic programming [5].

A *(monotonic) logic* is a pair $\mathcal{L} = \langle L, \vdash_{\mathcal{L}} \rangle$ where L is a set of formulas and $\vdash_{\mathcal{L}}$ is a Tarskian consequence relation over L, i.e., satisfying, for every $T \cup \Phi \cup \{\varphi\} \subseteq L$, ***Reflexivity:*** if $\varphi \in T$ then $T \vdash_{\mathcal{L}} \varphi$; ***Cut:*** if $T \vdash_{\mathcal{L}} \varphi$ for all $\varphi \in \Phi$, and $\Phi \vdash_{\mathcal{L}} \psi$ then $T \vdash_{\mathcal{L}} \psi$; ***Weakening:*** if $T \vdash_{\mathcal{L}} \varphi$ and $T \subseteq \Phi$ then $\Phi \vdash_{\mathcal{L}} \varphi$.

Let $Th(\mathcal{L})$ be the set of *theories of* $\mathcal{L}$, i.e. the set of subsets of L closed under the relation $\vdash_{\mathcal{L}}$. It is well-known that, for every (monotonic) logic $\mathcal{L}$, the tuple $\langle Th(\mathcal{L}), \subseteq \rangle$ is a complete lattice with smallest element the set $Theo = \{\varphi \in L : \vdash_{\mathcal{L}} \varphi\}$ of theorems of $\mathcal{L}$ and greatest element the set L of all formulas of $\mathcal{L}$.

In what follows we fix a (monotonic) logic $\mathcal{L} = \langle L, \vdash_{\mathcal{L}} \rangle$ and call it the *parameter logic*. The formulas of $\mathcal{L}$ are dubbed *(parametrized) atoms* and a *(parametrized) literal* is either a parametrized atom φ or its negation *not* φ, where as usual *not* denotes negation as failure. We dub *default literal* those of the form *not* φ. A *normal $\mathcal{L}$-parametrized logic program* is a set of rules of the form $\varphi \leftarrow \psi_1, \ldots, \psi_n, not\ \varphi_1, \ldots, not\ \varphi_m$ where $\varphi, \psi_1, \ldots, \psi_n, \varphi_1, \ldots, \varphi_m \in L$. A *definite $\mathcal{L}$-parametrized logic program* is a set of rules without negations as failure, i.e. of the form $\varphi \leftarrow \psi_1, \ldots, \psi_n$ where $\varphi, \psi_1, \ldots, \psi_n \in L$.

Let P_1 and P_2 be two normal $\mathcal{L}$-parametrized logic programs. We say that P_1 and P_2 are *strongly equivalent* if for every normal $\mathcal{L}$-parametrized logic program P, the programs $P_1 \cup P$ and $P_2 \cup P$ have the same stable models.

Given a parameter logic $\mathcal{L} = \langle L, \vdash \rangle$, we dub $HT_{\mathcal{L}}$ the $\mathcal{L}$-parametrized logic of Here-and-There. The language of $HT_{\mathcal{L}}$ is build constructively from the formulas of $\mathcal{L}$ using the connectives $\bot$, $\wedge$, $\vee$ and $\rightarrow$. Negation $\neg$ is introduced as an abbreviation $\neg\delta := (\delta \rightarrow \bot)$. The formulas of $\mathcal{L}$ act as atoms of $HT_{\mathcal{L}}$.

The semantics of $HT_{\mathcal{L}}$ is a generalization of the intuitionistic Kripke semantics of HT. A frame for $HT_{\mathcal{L}}$ is a tuple $\langle W, \leq \rangle$ where W is a set of exactly two worlds, say h (here) and t (there) with $h \leq t$. An $HT_{\mathcal{L}}$ interpretation is a frame together with an assignment i that associates to each world a theory of $\mathcal{L}$, such that $i(h) \subseteq i(t)$. Note the key idea of substituting, in the original definition of HT interpretations, sets of atoms by theories of the parameter logic. An interpretation is said to be *total* if $i(h) = i(t)$. It is convenient to see a $HT_{\mathcal{L}}$ interpretation as an ordered pair $\langle T^h, T^t \rangle$ such that $T^h = i(h)$ and $T^t = i(t)$ where i is the interpretation's assignment. We define the satisfaction relation between an $HT_{\mathcal{L}}$ interpretation $\langle T^h, T^t \rangle$ at a particular world w and a $HT_{\mathcal{L}}$ formula δ recursively, as follows:

i) for $\varphi \in L$ we have that $\langle T^h, T^t \rangle, w \Vdash \varphi$ if $T^w \vdash_{\mathcal{L}} \varphi$;
ii) $\langle T^h, T^t \rangle, w \nVdash \bot$;
iii) $\langle T^h, T^t \rangle, w \Vdash (\delta \vee \gamma)$ if $\langle T^h, T^t \rangle, w \Vdash \delta$ or $\langle T^h, T^t \rangle, w \Vdash \gamma$;
iv) $\langle T^h, T^t \rangle, w \Vdash (\delta \wedge \gamma)$ if $\langle T^h, T^t \rangle, w \Vdash \delta$ and $\langle T^h, T^t \rangle, w \Vdash \gamma$;
v) $\langle T^h, T^t \rangle, w \Vdash (\delta \to \gamma)$ if $\forall_{w' \geq w}$ we have $\langle T^h, T^t \rangle, w' \nVdash \delta$ or $\langle T^h, T^t \rangle, w' \Vdash \gamma$.

We say that an interpretation $\mathcal{I}$ is a *model* of an $HT_{\mathcal{L}}$ formula δ if $\mathcal{I}, w \Vdash \delta$ for every $w \in \{h, t\}$. A formula δ is said to be a consequence of a set of formulas Φ, denoted by $\Phi \vdash_{HT_{\mathcal{L}}} \delta$, if for every interpretation $\mathcal{I}$ and every world w we have that $\mathcal{I}, w \Vdash \delta$ whenever $\mathcal{I}, w \Vdash \delta'$ for every $\delta' \in \Phi$.

Definition 1. *An equilibrium model of a set Φ of $HT_{\mathcal{L}}$ formulas is a total $HT_{\mathcal{L}}$ interpretation $\langle T, T \rangle$ such that*

1. *$\langle T, T \rangle$ is a model of Φ;*
2. *for every $\mathcal{L}$ theory $T' \subset T$ we have that $\langle T', T \rangle$ is not a model of Φ.*

With this notion of model, equilibrium entailment is defined as follows.

Definition 2. *The equilibrium entailment, $\vDash_E$, over $HT_{\mathcal{L}}$ formulas is defined for every set $\Phi \cup \{\delta\}$ of $HT_{\mathcal{L}}$ formulas as follows:*

- *if Φ is non-empty and has equilibrium models then $\Phi \vDash_E \delta$ if every equilibrium model of Φ is an $HT_{\mathcal{L}}$ model of δ;*
- *if Φ is empty or does not have equilibrium models then $\Phi \vDash_E \delta$ if $\Phi \vdash_{HT_{\mathcal{L}}} \delta$.*

Clearly, the traditional HT logic is a particular case of our parametrized approach by taking as parameter logic $\mathcal{L} = \langle L, \vdash_{\mathcal{L}} \rangle$ where L is the set At of atoms of HT and $\vdash_{\mathcal{L}}$ is the only reasonable consequence definable over L, i.e., for every $X \cup \{p\} \subseteq At$ we have $X \vdash_{\mathcal{L}} p$ iff $p \in X$.

In what follows, we identify a rule $\varphi \leftarrow \psi_1, \ldots, \psi_n, not\ \varphi_1, \ldots, not\ \varphi_m$ of an $\mathcal{L}$-parametrized program with the $HT_{\mathcal{L}}$-formula $(\psi_1 \wedge \ldots \wedge \psi_n \wedge \neg\varphi_1 \wedge \ldots \wedge \neg\varphi_m) \to \varphi$, and a program P with the set of $HT_{\mathcal{L}}$-formulas that correspond to its rules. The following proposition states that parametrized equilibrium logic coincides with [5] in the specific case of logic programs.

Proposition 1. *For any $\mathcal{L}$-parametrized program P, an $HT_{\mathcal{L}}$ interpretation $\langle T, T \rangle$ is an equilibrium model of P iff T is stable model of P.*

The following theorem states that $HT_{\mathcal{L}}$ can be used to prove strong equivalence of parametrized logic programs.

Theorem 1. *Let P_1 and P_2 be two $\mathcal{L}$-parametrized logic programs. Then P_1 and P_2 are strongly equivalent iff P_1 and P_2 are equivalent in $HT_{\mathcal{L}}$.*

Note that, by construction, it is immediate that the semantics of general default logic given in [12] is a particular case of ours by taking CPL as parameter logic, HT_{CPL}. As a consequence, the strong connection proved in [12] between their semantics and the stable models like semantics of [11], also holds for HT_{CPL}. Concretely, they proved that if T is an CPL theory and Φ a set of GDL formulas then, $\langle T, T\rangle$ is an equilibrium model of Φ iff T is an extension (generalization of a stable model) of Φ.

3 Temporal Here-and-There Logic

Our parametrized logic is also interesting for parameters other than CPL. In particular, in this section we experiment with linear temporal logic LTL [9] and compare the obtained logic, HT_{LTL}, with the Temporal Here-and-There logic (THT) of [1].

The language of THT is built from the set $\mathcal{P}$ of propositional symbols using interchangeably HT connectives ($\bot, \to, \vee, \wedge$) and LTL temporal operators ($\bigcirc, \mathcal{U}, \mathcal{R}$). Negation is defined as $\neg\varphi \cong \varphi \to \bot$ whereas $\top \cong \neg\bot$. Other usual temporal operators can be defined using $\mathcal{U}$ and $\mathcal{R}$: $\Box\varphi \cong \bot\mathcal{R}\varphi$ and $\Diamond\varphi \cong \top\mathcal{U}\varphi$.

Recall that an LTL interpretation is a sequence $m = (m_i)_{i\in\mathbb{N}}$ where $m_i \subseteq \mathcal{P}$ for each $i \in \mathbb{N}$. A THT interpretation is a pair $M = \langle m^h, m^t\rangle$ where m^h, m^t are LTL interpretations such that $m^h_i \subseteq m^t_i$ for every $i \in \mathbb{N}$.

A THT interpretation has two dimensions. In the HT dimension we have the two worlds, h (here) and t (there) with $h \leq t$, and in the LTL dimension we have the time instants $i \in \mathbb{N}$. The satisfaction of a THT formula δ by a THT interpretation $M = \langle m^h, m^t\rangle$ is defined at a world $w \in \{h, t\}$ and at a time instant $i \in \mathbb{N}$, by structural induction:

- $M, w, i \not\Vdash \bot$;
- $M, w, i \Vdash p$ if $p \in m^w_i$, for $p \in \mathcal{P}$;
- $M, w, i \Vdash \varphi\mathcal{U}\psi$ if $\exists_{j\geq i}$ s.t. $M, w, j \Vdash \psi$ and $\forall_{i\leq k<j}$ we have $M, w, k \Vdash \varphi$;
- $M, w, i \Vdash \varphi\mathcal{R}\psi$ if $\forall_{j\geq i}$ either $M, w, j \Vdash \psi$ or $\exists_{i\leq k<j}$ s.t. $M, w, k \Vdash \varphi$;
- $M, w, i \Vdash (\varphi \vee \psi)$ if $M, w, i \Vdash \varphi$ or $M, w, i \Vdash \psi$;
- $M, w, i \Vdash (\varphi \wedge \psi)$ if $M, w, i \Vdash \varphi$ and $M, w, i \Vdash \psi$;
- $M, w, i \Vdash (\varphi \to \psi)$ if $\forall_{w'\geq w}$ we have $M, w', i \not\Vdash \varphi$ or $M, w', i \Vdash \psi$.

A THT interpretation M is a *model* of a THT formula φ, denoted by $M \Vdash \varphi$, if $M, h, 0 \Vdash \varphi$. This definition uses the so-called anchored version of LTL. Given a set Φ of THT formulas we denote by $Mod(\Phi)$ the set of THT interpretations that are models of every formula of Φ. A THT formula δ is *valid* if every THT interpretation is a model of δ. The consequence relation $\vdash_{THT}$ can be defined as $\Phi \vdash_{THT} \delta$ if for every interpretation M and world $w \in \{h, t\}$ we have that $M, w \Vdash \delta$ whenever $M, w \Vdash \delta'$ for every $\delta' \in \Phi$.

Recall that the language of THT is built using interchangeably HT connectives and LTL temporal operators. In HT_{LTL} (HT parametrized with LTL, as defined in Section 2) this interaction between the HT level and the parameter logic level is not allowed. For example, $\Box(p \to q)$ is not an HT_{LTL} formula. Although the language of THT is richer than that of HT_{LTL}, a result in [2] shows that every formula of THT as a normal form which is almost a formula of HT_{LTL}. In more detail, every THT formula is THT equivalent to a formula of the following form:

- an atom $p \in \mathcal{P}$;
- $\Box(B_1 \wedge \ldots \wedge B_n \to (C_1 \vee \ldots \vee C_m))$ where for each B_i and each C_j is a temporal literal, that is, of the form p, $\bigcirc p$ or $\neg p$ with $p \in \mathcal{P}$;
- $\Box(\Box p \to q)$ or $\Box(p \to \Diamond q)$ for some $p, q \in \mathcal{P}$.

By a normal temporal logic program we mean a set of rules of the form $\Box(\varphi \leftarrow \varphi_1, \ldots, \varphi_n, not\ \psi_1, \ldots, not\ \psi_m)$ where $\varphi, \varphi_1, \ldots, \varphi_n, \psi_1, \ldots, \psi_m$ are LTL^- formulas. By LTL^- we denote the LTL fragment of THT, i.e., LTL restricted to the temporal operators $(\bigcirc, \Box, \Diamond, \mathcal{U}, \mathcal{W}, \mathcal{R})$ (therefore, excluding classical negation and implication). We identify a rule $\Box(\varphi \leftarrow \varphi_1, \ldots, \varphi_n, not\ \psi_1, \ldots, not\ \psi_m)$ with the THT formula $\Box((\varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\psi_1 \wedge \ldots \wedge \neg\psi_m) \to \varphi)$. Note that this is not an HT_{LTL} formula. Let us now show how can we solve this mismatch.

Consider, for every LTL formula φ and every $i \in \mathbb{N}$, the LTL formula $\varphi^i := \bigcirc^i\varphi$, that is, φ preceded by i occurrences of $\bigcirc$. Let $r = \Box(\varphi \leftarrow \varphi_1, \ldots, \varphi_n, not\ \psi_1, \ldots, not\ \psi_m)$ be a normal temporal logic program rule and consider the set $r^* = \{\varphi^i \leftarrow \varphi_1^i, \ldots, \varphi_n^i, not\ \psi_1^i, \ldots, not\ \psi_m^i : i \in \mathbb{N}\}$ of LTL-parametrized rules obtained from r. Given a normal temporal logic program P we consider the LTL-parametrized normal logic program $P^* = \bigcup_{r \in P} r^*$.

The following theorem relates logical equivalence in THT and in HT_{LTL}.

Theorem 2. *Let P_1 and P_2 be normal temporal logic programs. Then, P_1 and P_2 are logically equivalent in THT iff P_1^* and P_2^* are logically equivalent in HT_{LTL}.*

Note that, contrarily to THT, logical equivalence in HT_{LTL} is a necessary and sufficient condition for strong equivalence of temporal logic programs. Note also that HT_{LTL} contains classical negation (usually called explicit negation) and classical implication. This is clearly a plus in many reasoning scenarios.

4 Conclusions

We have defined a parametrized version of equilibrium logic and of its monotone base, the logic of Here-and-There, by allowing complex formulas of a parameter logic as atoms. We proved that both equilibrium logic and HT are particular cases of our approach and, moreover, we generalized the relation between equilibrium logic and the stable model semantics for logic programs. In particular we proved that logical equivalence in parametrized HT captures strong equivalence of parametrized logic programs. By taking classical logic as parameter logic, we proved that general default logic is a particular case of our approach. We ended with an example where linear temporal logic was taken as parameter logic, thus allowing to characterize strong equivalence of temporal logic programs.

The work raises several interesting paths for future work. One that is already ongoing is to define a parametrized version of partial equilibrium logic and generalise its relation to partial stable model semantics of logic programs. It would be interesting to find an axiomatization of parametrized HT. In order to access its merits in full, other interesting examples of parameter logic should be studied. A very interesting example is the case of first-order logic. We intend to compare the resulting parametrized equilibrium logic with the first-order version of equilibrium logic [8].

References

1. Aguado, F., Cabalar, P., Pérez, G., Vidal, C.: Strongly equivalent temporal logic programs. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 8–20. Springer, Heidelberg (2008)
2. Cabalar, P.: A normal form for linear temporal equilibrium logic. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 64–76. Springer, Heidelberg (2010)
3. Ferraris, P.: Answer sets for propositional theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 119–131. Springer, Heidelberg (2005)
4. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080 (1988)
5. Gonçalves, R., Alferes, J.: Parametrized logic programming. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 182–194. Springer, Heidelberg (2010)
6. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Trans. Comput. Log. 2(4), 526–541 (2001)
7. Pearce, D.: Equilibrium logic. Ann. Math. Artif. Intell. 47(1-2), 3–41 (2006)
8. Pearce, D., Valverde, A.: Towards a first order equilibrium logic for nonmonotonic reasoning. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 147–160. Springer, Heidelberg (2004)
9. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, R.I., pp. 46–57. IEEE Comput. Sci., Long Beach (1977)
10. Reiter, R.: A logic for default reasoning. Artificial Intelligence (13) (1980)
11. Zhou, Y., Lin, F., Zhang, Y.: General default logic. Ann. Math. Artif. Intell. 57(2), 125–160 (2009)
12. Zhou, Y., Zhang, Y.: Rule calculus: Semantics, axioms and applications. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 416–428. Springer, Heidelberg (2008)

Random vs. Structure-Based Testing of Answer-Set Programs: An Experimental Comparison*

Tomi Janhunen[1], Ilkka Niemelä[1], Johannes Oetsch[2], Jörg Pührer[2], and Hans Tompits[2]

[1] Aalto University, Department of Information and Computer Science,
P.O. Box 15400, FI-00076 Aalto, Finland
{Tomi.Janhunen,Ilkka.Niemela}@aalto.fi

[2] Technische Universität Wien, Institut für Informationssysteme 184/3,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. Answer-set programming (ASP) is an established paradigm for declarative problem solving, yet comparably little work on testing of answer-set programs has been done so far. In a recent paper, foundations for structure-based testing of answer-set programs building on a number of coverage notions have been proposed. In this paper, we develop a framework for testing answer-set programs based on this work and study how good the structure-based approach to test input generation is compared to random test input generation. The results indicate that random testing is quite ineffective for some benchmarks, while structure-based techniques catch faults with a high rate more consistently also in these cases.

Keywords: answer-set programming, structure-based testing, random testing.

1 Introduction

Answer-set programming (ASP) is an established approach to declarative problem solving with increasingly efficient solver technology enabling applications in a large range of areas. However, comparably little effort has been spent on software development methods and, in particular, on developing testing techniques for answer-set programs.

In previous work [1], we introduced a *structure-based testing approach* for answer-set programs. In this approach, testing is based on test cases that are chosen with respect to the internal structure of a given answer-set program. More specifically, we introduced different notions of *coverage* that measure to what extent a collection of test inputs covers certain important structural components of a program. Another typical method used in conventional software development is *random testing*. Although random testing is able to discover some errors quite quickly, it often needs to be complemented with other techniques to increase test coverage. The effectiveness of such more systematic testing methods is usually evaluated by comparison to random testing [2].

In this paper, we continue our work on testing and develop methods for *structure-based test input generation* based on our previous work [1] and for *random test input generation*. The basic research question we address is how the structure-based approach

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698 and the Academy of Finland under project 122399.

to test input generation compares to random test input generation. To this end, we evaluate structure-based and random testing using benchmark problems from the second ASP competition [3]. The evaluation is based on *mutation analysis* [4], i.e., program instances are rendered incorrect by injecting faults according to a mutation model. Then, the two approaches are compared on the basis how well they can catch such faults.

2 Background

We deal with a class of logic programs that corresponds to the ground fragment of the `Gringo` input language [5,6], which extends normal programs by *aggregate atoms*. The semantics of this class is defined in terms of *answer sets* (also known as *stable models*) [7]. We denote the collection of all answer sets of a program P by $\mathrm{AS}(P)$. Given a program P and an interpretation I, the set of *supporting rules of P with respect to I*, $\mathrm{SuppR}(P, I)$, consists of all rules $r \in P$ whose body is true in I. Moreover, for an ordinary atom a, $\mathrm{Def}_P(a)$ is the set of *defining rules* of a in P, i.e., the set of all rules in P where a occurs positively in the rule head (possibly within an aggregate atom).

We next review some definitions related to testing logic programs [1]. With each program P, we associate two finite sets of ordinary atoms, $\mathbb{I}_P$ and $\mathbb{O}_P$, which are the respective input and output signatures of P. A *test input*, or simply *input*, of P is a finite set of atoms from $\mathbb{I}_P$. Given some input I of P, the *output* of P with respect to I, $P[I]$, is defined as $P[I] = \{S \cap \mathbb{O}_P \mid S \in \mathrm{AS}(P \cup I)\}$. Moreover, a *specification* s for P is a mapping from the set of test inputs of P into sets of subsets of $\mathbb{O}_P$. P is considered to be *correct* with respect to its specification s if $P[I] = s(I)$ for each test input I of P. Moreover, we define (i) a *test case* for P as a pair $\langle I, O\rangle$, where I is an input of P and $O = s(I)$, (ii) a *test suite* for P as a set of test cases for P, and (iii) the *exhaustive test suite* for P as the collection of all test cases for P. We say that P *passes* a test case $T = \langle I, O\rangle$ if $P[I] = O$, and P *fails* for T otherwise; and P passes a test suite $\mathcal{S}$ if P passes all test cases in $\mathcal{S}$, and P fails for $\mathcal{S}$ otherwise.

Given a program P and an input I of P, a proper rule $r \in P$ (i.e., a rule which is not a constraint) is *positively covered* (resp., *negatively covered*) by I if $r \in \mathrm{SuppR}(P, X)$ (resp., $r \notin \mathrm{SuppR}(P, X)$) for some $X \in \mathrm{AS}(P \cup I)$. A non-empty definition $\mathrm{Def}_P(a)$ of an ordinary atom a appearing in P is *positively covered* (resp., *negatively covered*) by I if there is some $X \in \mathrm{AS}(P \cup I)$ such that $a \in X$ (resp., $a \notin X$). On the other hand, a constraint $c \in P$ is *covered* by I if the body of c is true in some answer set $X \in \mathrm{AS}((P \setminus \{c\}) \cup I)$, i.e., c is temporarily removed from P for this notion. For any of these coverage notions, we say that a test suite $\mathcal{S}$ yields *total coverage* on a program P if each rule (definition, constraint) in P which is covered (positively and negatively) by an input of the exhaustive test suite for P is covered by an input from $\mathcal{S}$.

3 Experimental Setup

Our experimental setup involves six steps: (1) selecting some benchmark programs; (2) grounding them to obtain reference programs; (3) injecting faults into the reference programs to generate mutants; (4) generating random test suites for mutants; (5) generating structure-based test suites for mutants; and (6) comparing the resulting test suites. We describe these steps in more detail in what follows.

(*1*) *Selection of Benchmark Instances.* We consider programs from the benchmark collection of the second ASP competition [3] for our experiments. In particular, we selected the problems *Maze Generation*, *Solitaire*, *Graph Partitioning*, and *Reachability* to represent typical program structures in ASP. The Maze Generation problem is about generating a maze on a two-dimensional grid. The problem encoding is prototypical for the guess-and-check paradigm in ASP and involves inductive definitions to express the reachability conditions. A distinguishing feature of this problem is that solutions are partially fixed in the input and completed in the output. Solitaire is a simple planning game that is played on a grid with 33 cells. The structure of the encoding is slightly simpler than for Maze Generation since no positive recursion is involved. Graph Partitioning is a graph-theoretical problem about partitioning the nodes of a weighted graph subject to certain conditions. It is representative for hard problems on linked data structures that involve integer calculations. Finally, the objective of the Reachability problem is to exhibit a path between two dedicated nodes in a directed graph. Reachability is a representative for problems solvable in polynomial time. The central aspect is to efficiently compute the transitive closure of the edge relation of the input graph.

(*2*) *Grounding.* Since our testing approach is defined for propositional programs, we need to ground the programs from Step (1). Uniform ASP encodings usually involve a natural parameter that allows one to scale the size of ground instances. For Maze Generation, we used a bound on the size of the grid of 7×7 to obtain a finite grounding. Since Solitaire is a planning problem, we parameterized it by setting the upper bound on the lengths of plans to 4. For Graph Partitioning, since it takes graphs as input, one natural scaling parameter is a bound on the number of nodes which we fixed to 7. Furthermore, we fixed a bound of 3 on the maximal number of partitions and a bound of 20 on maximal weights assigned to edges. For Reachability, we fixed an upper bound of 6 on the number of nodes as the scaling parameter. We refer to the ground program obtained in this step as *reference program* since it later serves as a *test oracle*, i.e., as a method to determine the correct output given some input.

(*3*) *Fault Injection.* Based on grounded encodings from the ASP competition, we follow an approach that is inspired by *mutation analysis* [4]. In this step, we apply certain *mutation operations* on a program to obtain versions, so called *mutants*, where small faults have been injected. Intuitively, mutation operations mimic typical mistakes made by a programmer like omission of certain elements or misspelling of names. In particular, we generate different classes of mutants, one class for each mutation operation. To this end, we consider (i) deleting a single body literal, (ii) deleting a single rule of a program, (iii) swapping the polarity of literals, (iv) increasing or decreasing the bound of an aggregate by one, and (v) replacing an atom by another atom appearing in the program. For each grounded benchmark encoding, we generated different classes of mutants, in particular, one class of mutants for each mutation operation. The bound modification was not applied to the Maze Generation and Reachability instances which do not involve aggregate atoms nor bounds. Each mutant was generated by applying a mutation operation precisely once. Each class consists of 100 mutants which are publicly available[1]. We used dedicated equivalence tests [8] to eliminate equivalent mutants that cannot be distinguished from the reference program by any test input.

[1] `http://www.kr.tuwien.ac.at/research/projects/mmdasp/mutant.tgz`

(*4*) *Random Test Suite.* In software testing, a rigorous theoretical analysis of the fault detection capabilities of systematic testing methods is practically infeasible. What usually is done is that systematic methods are compared to the *random-testing standard* [2]. Random testing means that test inputs are randomly selected from the input space—usually a uniform distribution is used. For each benchmark problem in our experiments, we formalized the preconditions as a simple program mainly consisting of integrity constraints. Any model of such a program constitutes an admissible input, i.e., one which is consistent with the preconditions of the encoding. Hence, the task of generating admissible random inputs can be reduced to computing uniformly-distributed answer sets of logic programs.

Our approach to random testing is to incrementally and in a highly randomized way build an answer set of the program that formalizes the preconditions of a benchmark problem. The algorithm takes a program P with input signature $\mathbb{I}_P$ and generates a random test input I for P such that $P[I] \neq \emptyset$ incrementally starting from the empty set. For each atom from $\mathbb{I}_P$, one after the other, we decide by tossing a coin whether or not it should be contained in the final input. Furthermore, we always check if such a decision allows to proceed in a way that the final input is consistent with the program. This check is realized using ASP and a method which does not guarantee strictly uniform distribution of inputs as a reasonable compromise. For each mutant of each benchmark class, we generated 1000 random inputs.

(*5*) *Structure-Based Test Suite.* We next review our approach to generate test inputs that yield total coverage. First, we sketch how we can use ASP itself to generate covering inputs. Let P be a program with input signature $\mathbb{I}_P$. We define the *input generator* for P, denoted $\mathrm{IG}(P)$, as the program $\{a' \leftarrow \mathrm{not}\ a''; a'' \leftarrow \mathrm{not}\ a'; a \leftarrow a' \mid a \in \mathbb{I}_P\}$ where all primed and double primed atoms do not occur anywhere else. A program P is labeled for rule coverage as the program $\mathrm{LR}(P) = \{\mathrm{H}(r) \leftarrow r'; r' \leftarrow \mathrm{B}(r) \mid r \in P\}$ where r' is a globally new label for each $r \in P$. To cover individual rules either positively or negatively, we use programs $P_r^+ = \mathrm{LR}(P) \cup \{\leftarrow \mathrm{not}\ r'\}$ and $P_r^- = \mathrm{LR}(P) \cup \{\leftarrow r'\}$, respectively. Then, the inputs for P that cover r positively and negatively are in a one-to-one correspondence with the respective sets $\{X \cap \mathbb{I}_P \mid X \in \mathrm{AS}(\mathrm{IG}(P) \cup P_r^+)\}$ and $\{X \cap \mathbb{I}_P \mid X \in \mathrm{AS}(\mathrm{IG}(P) \cup P_r^-)\}$ of inputs. Similar reductions can be devised for the other coverage notions. The preconditions of each benchmark program P form a set C of constraints which can be incorporated to the preceding reductions in this way restricting the reductions to admissible test inputs of P. Given a mutated program P, we generate test suites that yield total rule coverage for P using the reduction P_r^+. We first generate a test suite $\mathcal{S}$ that obtains total positive rule coverage for P. To this end, we pick a rule $r \in P$ not yet positively covered and check whether there is an input and a corresponding answer set X that positively covers r. If such an input exists, r is marked as positively covered, if not, r is marked as positively failed, meaning that no inputs exist that cover r positively. Then, some simple bookkeeping takes place: any rule that is positively, resp., negatively, covered by the answer set X is marked accordingly. The procedure iterates until all rules are marked as positively covered or failed. A similar procedure is then applied to extend $\mathcal{S}$ to obtain total negative rule coverage. The method for the other notions of coverage is analogous. Note that for a program P, there is a test suite that obtains total rule, definition, or constraint coverage whose

Table 1. Catching rates for the Maze Generation and Reachability benchmark

Maze Generation

	size of test suite	AR	LD	RD	PS	Total
Random Testing	1000	0.03	0.18	0.00	0.16	0.09
Definition Coverage	36	0.67	0.63	0.74	0.78	0.71
Rule Coverage	85	0.78	0.66	0.78	0.75	0.74
Constraint Coverage	176	0.81	0.74	0.81	0.90	0.82
Definition & Constraint Coverage	212	0.86	0.87	0.85	0.93	0.88
Rule & Constraint Coverage	261	0.89	0.88	0.86	0.94	0.89

Reachability

	size of test suite	AR	LD	RD	PS	Total
Random Testing	1000	0.61	0.56	0.59	0.69	0.61
Definition Coverage	37	0.09	0.17	0.00	0.01	0.07
Rule Coverage	592	0.47	0.67	0.07	0.63	0.46
Constraint Coverage	36	0.15	0.02	0.10	0.07	0.09
Definition & Constraint Coverage	73	0.21	0.19	0.10	0.08	0.15
Rule & Constraint Coverage	628	0.56	0.69	0.17	0.64	0.52

size is bounded by the number of rules in P. Though our method does not guarantee minimality of a test suite, it always produces one within this bound.

(*6*) *Comparison.* Finally, the different classes of test inputs are evaluated with respect to their potential to *catch* mutants, i.e., to reveal the injected faults. To determine whether a mutant passes a test case, we use the reference program as a test oracle. Given a class M of mutants and a test suite $\mathcal{S}$, the catching rate of $\mathcal{S}$ on M is the ratio of mutants in M that fail for $\mathcal{S}$ and the total number of mutants in M.

4 Results and Discussion

Our test results in terms of catching rates for the Maze Generation and Reachability benchmarks are summarized in Table 1. The mutation operations studied were atom replacement (AR), literal deletion (LD), rule deletion (RD), and literal polarity swapping (PS). Besides definition, rule, and constraint coverage, we considered the combinations of definition and rule coverage with constraint coverage. The column "size of test suite" in the tables refers to the size of the generated test suites for each mutant class. For the coverage-based suites, the numbers are averages over all test suites in a class.

For Maze Generation, random testing only obtains very low catching rates, while systematic testing yields total rates of up to 0.9 using a significantly smaller test suite. One explanation for the poor performance of random testing is that the probability that a random test input is consistent with a mutant is very low. Inputs that yield no or few outputs for a mutant seem to have a low potential for fault detection. The situation is different for the Reachability benchmark: random testing performs slightly better than systematic testing. This is mainly explained by the higher number of test inputs used for random testing. The conclusion is that the considered structural information seems to provide no additional advantage compared to randomly picking inputs for this benchmark. As regards structural testing, the best detection rate is obtained by combining rule and constraint coverage. However, this comes at a cost of larger test suites compared to the other coverage notions. We conducted similar experiments using the Graph Partitioning and Solitaire benchmarks. Interestingly, it turned out that testing seems to be trivial in these cases. Since both random and structural testing achieved a constant catching rate of 1.00 for all mutant classes, we omit the respective tables. In fact, almost any test input catches all mutants for these programs. This nicely illustrates how the non-determinism involved in answer-set semantics affects testing. While in conventional testing only a single execution results from a particular test input, the situation in ASP is different. Given a test input, the output of a uniform encoding can consist of a vast number of answer sets, each of which potentially revealing a fault. For example, typical random inputs for Solitaire yield tens of thousands of answer sets which gives quite a lot information for testing. In other words, the probability that some effects of a fault show up in at least one answer set seems to be quite high.

For some encodings, systematic testing gives a clear advantage over random testing while for other programs, it could not be established that one approach is better than the other. These findings are in principle consistent with experiences in conventional software testing. The results indicate that random testing is quite effective in catching errors provided that sufficiently many admissible test inputs are considered. There is no clear-cut rule when to use random or systematic testing. The advantage of random testing is that inputs can be easily generated, directly based on a specification, and that the fault detection rates can be surprisingly good. The advantage of structure-based testing is that resulting test suites tend to be smaller which is especially important if testing is expensive. For some problems, structure-based testing is able to detect faults at a high rate but random testing is very ineffective even with much larger test suites.

References

1. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proc. ECAI 2010, pp. 951–956. IOS Press, Amsterdam (2010)
2. Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering, pp. 970–978. Wiley, Chichester (1994)
3. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
4. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection help for the practicing programmer. IEEE Computer 11(4), 34–41 (1978)
5. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
6. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder gringo. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 502–508. Springer, Heidelberg (2009)
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. 5th Logic Programming Symposium, pp. 1070–1080. MIT Press, Cambridge (1988)
8. Janhunen, T., Oikarinen, E.: LPEQ and DLPEQ – translators for automated equivalence testing of logic programs. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 336–340. Springer, Heidelberg (2003)

Integrating Rules and Ontologies in the First-Order Stable Model Semantics (Preliminary Report)

Joohyung Lee and Ravi Palla

School of Computing, Informatics and Decision Systems Engineering
Arizona State University
Tempe, AZ, 85287, USA
{joolee,Ravi.Palla}@asu.edu

Abstract. We present an approach to integrating rules and ontologies on the basis of the first-order stable model semantics defined by Ferraris, Lee and Lifschitz. We show that a few existing integration proposals can be uniformly related to the first-order stable model semantics.

1 Introduction

Integrating nonmonotonic rules and ontologies has received much attention, especially in the context of the Semantic Web. A *hybrid knowledge base (hybrid KB)* is a pair $(\mathcal{T}, \mathcal{P})$ where $\mathcal{T}$ is a first-order logic (FOL) knowledge base (typically in a description logic (DL)) and $\mathcal{P}$ is a logic program. The existing integration approaches can be classified into three categories [1]. In the *loose integration* approach (e.g., [1]), $\mathcal{T}$ is viewed as an external source of information with its own semantics that can be accessed by entailment-based query interfaces from $\mathcal{P}$. In the *tight integration with semantic separation* approach (e.g.,[2; 3; 4]), the semantics of logic programs are adapted to allow predicates of $\mathcal{T}$ in the rules, thereby leading to a more tight coupling. On the other hand, a model of the hybrid KB is constructed by the union of a model of $\mathcal{T}$ and a model of $\mathcal{P}$. In the *tight integration under a unifying logic* approach (e.g.,[5; 6]), $\mathcal{T}$ and $\mathcal{P}$ are treated uniformly as they are embedded into a unifying nonmonotonic logic.

Typically, existing integration approaches assume that the underlying signature does not contain function constants of positive arity. We represent the signature by $\langle C, P\rangle$ where C is a set of object constants and P is a set of predicate constants. Formally, a hybrid KB $(\mathcal{T}, \mathcal{P})$ of the signature $\langle C, P_{\mathcal{T}} \cup P_{\mathcal{P}}\rangle$ where $P_{\mathcal{T}}$ and $P_{\mathcal{P}}$ are disjoint sets of predicate constants, consists of a first-order logic knowledge base $\mathcal{T}$ of signature $\langle C, P_{\mathcal{T}}\rangle$ and a logic program $\mathcal{P}$ of signature $\langle C, P_{\mathcal{T}} \cup P_{\mathcal{P}}\rangle$.

In this paper, we investigate whether the first-order stable model semantics (FOSM) [7], which naturally extends both first-order logic and logic programs, can serve as a unifying logic for the integration of rules and ontologies. As the first step, we show how some of the well-known integration proposals from each category, namely, nonmonotonic dl-programs [1] (loose integration), $\mathcal{DL} + \mathit{log}$

[3] (tight integration with semantic separation), and quantified equilibrium logic based integration [5] (tight integration under a unifying logic), can be related to the first-order stable model semantics.

2 FOSM Based Hybrid KB

We refer the reader to [7] for the definition of the first-order stable model semantics, which applies to any first-order sentence. There the stable models of a first-order sentence F relative to a list $\mathbf{p}$ of predicates are defined as the models of the second-order sentence $\mathrm{SM}[F;\mathbf{p}]$ (in the sense of classical logic). Syntactically, $\mathrm{SM}[F;\mathbf{p}]$ is the formula

$$F \wedge \neg\exists\mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})), \tag{1}$$

where $\mathbf{u}$ is a list of predicate variables corresponding to $\mathbf{p}$, and F^* is defined recursively (See [7] for the details). In general, $\mathbf{p}$ is any list of predicate constants called *intensional* predicates—the predicates that we "intend to characterize" by F. Logic programs are identified as a special class of first-order theories by turning them into their *FOL-representations*. In [7], it is shown that the answer sets of a logic program $\mathcal{P}$ are precisely the Herbrand interpretations that satisfy $\mathrm{SM}[F;\mathbf{p}]$, where F is the FOL-representation of $\mathcal{P}$ and $\mathbf{p}$ is the list of all predicate constants occurring in $\mathcal{P}$. In another special case when $\mathbf{p}$ is empty, $\mathrm{SM}[F;\mathbf{p}]$ is equivalent to F. Consequently, both logic programs and first-order logic formulas can be viewed as special cases of $\mathrm{SM}[F;\mathbf{p}]$ depending on the choice of intensional predicates $\mathbf{p}$. As we show below, the distinction between intensional and non-intensional predicates is useful in characterizing hybrid KBs.

Throughout this paper, we assume that a hybrid KB contains finitely many rules in $\mathcal{P}$[1]. We identify a hybrid KB $(\mathcal{T},\mathcal{P})$ of signature $\langle C, P_{\mathcal{T}} \cup P_{\mathcal{P}}\rangle$ with the second-order sentence $\mathrm{SM}[FO(\mathcal{T}) \wedge FO(\mathcal{P}); P_{\mathcal{P}}]$ of the same signature, where $FO(\mathcal{T})$ ($FO(\mathcal{P})$, respectively) is the first-order logic (FOL) representation of $\mathcal{T}$ ($\mathcal{P}$, respectively).

Example 1. [5, Example 1] Consider a hybrid KB consisting of a first-order logic theory $\mathcal{T}$

$$\forall x(PERSON(x) \rightarrow (AGENT(x) \wedge (\exists y HAS\text{-}MOTHER(x,y))))$$
$$\forall x((\exists y HAS\text{-}MOTHER(x,y)) \rightarrow ANIMAL(x))$$

(every $PERSON$ is an $AGENT$ and has some (unknown) mother, and everyone who has a mother is an $ANIMAL$) and a nonmonotonic logic program $\mathcal{P}$

$$PERSON(x) \leftarrow AGENT(x), not\ machine(x)$$
$$AGENT(DaveB)$$

[1] This is for simplicity of applying SM. Alternatively we may extend SM to (possibly infinite) sets of formulas.

($AGENT$s are by default $PERSON$s, unless known to be $machine$s, and $DaveB$ is an $AGENT$). Here $P_{\mathcal{T}}$ is $\{PERSON, AGENT, HAS\text{-}MOTHER, ANIMAL\}$, and $P_{\mathcal{P}}$ is $\{machine\}$. Formula $\mathrm{SM}[FO(\mathcal{T}) \wedge FO(\mathcal{P});\ machine]$ entails $PERSON(DaveB)$. Furthermore, it entails each of $\exists y HAS\text{-}MOTHER(DaveB, y)$ and $ANIMAL(DaveB)$.

In fact, this treatment of a hybrid KB is essentially equivalent to the quantified equilibrium logic (QEL) based approach, as stated in Theorem 15 from [5]. The equivalence is also immediate from Lemma 9 from [7], which shows the equivalence between the first-order stable model semantics and QEL. de Bruijn *et al.* [5] show that a few other integration approaches, such as r-hybrid, r^+-hybrid, and g-hybrid KBs, can be embedded into QEL-based hybrid KBs. Consequently, they can also be represented by the first-order stable model semantics.

In the following we relate $\mathcal{DL}+log$ [3] and nonmonotonic dl-programs [1] to the first-order stable model semantics.

3 Relating to $\mathcal{DL}+log$ by Rosati

We refer the reader to [3] for the nonmonotonic semantics of $\mathcal{DL}+log$. A $\mathcal{DL}+log$ knowledge base is $(\mathcal{T}, \mathcal{P})$ where $\mathcal{T}$ is a DL knowledge base of signature $\langle C, P_{\mathcal{T}}\rangle$ and $\mathcal{P}$ is a (disjunctive) Datalog program of signature $\langle C, P_{\mathcal{T}} \cup P_{\mathcal{P}}\rangle$. $\mathcal{DL}+log$ imposes the *standard name assumption*: every interpretation is over the same fixed, countably infinite domain Δ, and in addition the set C of object constants is such that it is in the same one-to-one correspondence with Δ in every interpretation. As a result, for simplicity, we identify Δ with C.

In $\mathcal{DL}+log$, the predicates from $P_{\mathcal{T}}$ are not allowed to occur in the negative body of a rule in $\mathcal{P}$. In order to ensure decidable reasoning, $\mathcal{DL}+log$ imposes two conditions: *Datalog safety* and *weak safety*. The rules of $\mathcal{P}$ are called *Datalog safe* if every variable occurring in a rule also occurs in the positive body of the rule, and they are called *weakly safe* if every variable occurring in the head of a rule also occurs in a Datalog atom in the positive body of the rule.

The nonmonotonic semantics of $\mathcal{DL}+log$ is based on the stable model semantics for disjunctive logic programs. The following proposition shows how the nonmonotonic semantics of $\mathcal{DL}+log$ can be reformulated in terms of the first-order stable model semantics.

Proposition 1. *For any $\mathcal{DL}+log$ knowledge base $(\mathcal{T}, \mathcal{P})$, under the standard name assumption, the nonmonotonic models of $(\mathcal{T}, \mathcal{P})$ according to [3] are precisely the interpretations of $\langle C, P_{\mathcal{T}} \cup P_{\mathcal{P}}\rangle$ that satisfy* $\mathrm{SM}[FO(\mathcal{T}) \wedge FO(\mathcal{P});\ P_{\mathcal{P}}]$.

Since the reformulation does not refer to grounding, arguably, it provides a simpler account of $\mathcal{DL}+log$ in comparison with the original semantics in [3].

In view of the relationship between the two formalisms in Proposition 1, we observe that the condition of weak safety imposed in $\mathcal{DL}+log$ coincides with the condition of *semi-safety* from [8] that applies to $FO(\mathcal{T}) \wedge FO(\mathcal{P})$ when we take

$P_\mathcal{P}$ as intensional predicates[2]. Using the results on semi-safety presented in [8], below we show that the requirement of Datalog safety can be dropped without affecting the decidability of reasoning in $\mathcal{DL}+log$.

Proposition 2. *Let $\mathcal{K} = (\mathcal{T}, \mathcal{P})$ be a $\mathcal{DL}+log$ knowledge base such that $\mathcal{P}$ is weakly safe but is not necessarily Datalog safe. Let $\mathcal{P}'$ be the program obtained from $\mathcal{P}$ by removing in every rule, all the negative Datalog literals that contain a variable that occurs only in the negative body. Then $\mathcal{K}$ is equivalent (under the nonmonotonic semantics) to the $\mathcal{DL}+log$ knowledge base $(\mathcal{T}, \mathcal{P}')$.*

Since the complexity of the transformation required to obtain $\mathcal{P}'$ is polynomial in the size of $\mathcal{P}$, Proposition 2 tells us that the decidability results (Theorems 11 and 12 from [3]) and the complexity results (Theorem 13 from [3]) with respect to the nonmonotonic semantics of $\mathcal{DL}+log$ can be straightforwardly carried over to $\mathcal{DL}+log$ knowledge bases $(\mathcal{T}, \mathcal{P})$ where $\mathcal{P}$ is weakly safe but not necessarily Datalog safe.

4 Relating to Nonmonotonic dl-Programs by Eiter *et al.*

A nonmonotonic *dl-program* [1] is a pair $(\mathcal{T}, \mathcal{P})$, where $\mathcal{T}$ is a DL knowledge base of signature $\langle C, P_\mathcal{T}\rangle$ and $\mathcal{P}$ is a *generalized* normal logic program of signature $\langle C, P_\mathcal{P}\rangle$ such that $P_\mathcal{T} \cap P_\mathcal{P} = \emptyset$. A generalized normal logic program is a set of nondisjunctive rules that can contain queries to $\mathcal{T}$ in the form of "dl-atoms." A *dl-atom* is of the form

$$DL[S_1 op_1 p_1, \ldots, S_m op_m p_m;\ Q](\mathbf{t}) \quad (m \geq 0) \tag{2}$$

where $S_i \in P_\mathcal{T}$, $p_i \in P_\mathcal{P}$, and $op_i \in \{\oplus, \odot, \ominus\}$; $Q(\mathbf{t})$ is a *dl-query* [1].

The semantics of dl-programs is defined by extending the answer set semantics to generalized programs. For this, the definition of satisfaction is extended to ground dl-atoms. An Herbrand interpretation I *satisfies* a ground atom A *relative to* $\mathcal{T}$ if I satisfies A. An Herbrand interpretation I *satisfies* a ground dl-atom (2) *relative to* $\mathcal{T}$ if $\mathcal{T} \cup \bigcup_{i=1}^m A_i(I)$ entails $Q(\mathbf{t})$, where $A_i(I)$ is

- $\{S_i(\mathbf{e}) \mid\ p_i(\mathbf{e}) \in I\}$ if op_i is $\oplus$,
- $\{\neg S_i(\mathbf{e}) \mid\ p_i(\mathbf{e}) \in I\}$ if op_i is $\odot$,
- $\{\neg S_i(\mathbf{e}) \mid\ p_i(\mathbf{e}) \notin I\}$ if op_i is $\ominus$,

The satisfaction relation is extended to allow propositional connectives in the usual way.

Eiter *et al.* [1] define two semantics of dl-programs, which are based on different definitions of a reduct. In defining *weak* answer sets, the reduct is obtained from the given program by eliminating all dl-atoms (similar to the way that the

[2] The definition of semi-safety (called "argument-restricted" in that paper) is more general. That definition applies to any prenex formula even allowing function constants of positive arity.

negative literals in the body are eliminated in forming the reduct). In defining *strong* answer sets, the reduct is obtained from the given program by eliminating all nonmonotonic dl-atoms, but leaving monotonic dl-atoms. Below we show that each semantics can be characterized by our approach by extending F^* to handle dl-atoms in different ways.

For this, we define *dl-formulas* of signature $\langle C, P_\mathcal{T} \cup P_\mathcal{P}\rangle$ as an extension of first-order formulas by treating dl-atoms as a base case in addition to standard atomic formulas formed from $\langle C, P_\mathcal{P}\rangle$[3]. Note that any generalized normal logic program can be viewed as a dl-formula: $FO(\mathcal{P})$ can be extended to a generalized normal logic program $\mathcal{P}$ in a straightforward way. Let F be a ground dl-formula[4]. We define F^{w*} the same as F^* except for a new clause for a dl-atom:

$$DL[S_1 op_1 p_1, \dots, S_m op_m p_m; Q](\mathbf{c})^{w*}(\mathbf{u}) = DL[S_1 op_1 p_1, \dots, S_m op_m p_m; Q](\mathbf{c}).$$

$\mathrm{SM}^w[F]$ is defined the same as formula (1) except that F^{w*} is used in place of F^*. The following proposition shows how weak answer sets can be characterized by this extension. The notion $FO(\mathcal{P})$ is straightforwardly extended to a generalized normal logic program by treating dl-atoms like standard atoms.

Proposition 3. *For any dl-program $(\mathcal{T}, \mathcal{P})$ such that $\mathcal{P}$ is ground, the weak answer sets of $(\mathcal{T}, \mathcal{P})$ are precisely the Herbrand interpretations of signature $\langle C, P_\mathcal{P}\rangle$ that satisfy $\mathrm{SM}^w[FO(\mathcal{P});\ P_\mathcal{P}]$ relative to $\mathcal{T}$.*

In order to capture strong answer sets, we define F^{s*} the same as F^* except for a new clause for a dl-atom:

$$DL[S_1 op_1 p_1, \dots, S_m op_m p_m; Q](\mathbf{c})^{s*}(\mathbf{u}) = DL[S_1 op_1 u_1, \dots, S_m op_m u_m; Q](\mathbf{c})$$

($u_1, \dots, u_m$ are the elements of $\mathbf{u}$ that correspond to $p_1, \dots, p_m$) if this dl-atom is monotonic; otherwise

$$DL[S_1 op_1 p_1, \dots, S_m op_m p_m; Q](\mathbf{c})^{s*}(\mathbf{u}) = DL[S_1 op_1 p_1, \dots, S_m op_m p_m; Q](\mathbf{c}).$$

$\mathrm{SM}^s[F]$ is defined the same as formula (1) except that F^{s*} is used in place of F^*. The following proposition shows how strong answer sets can be characterized by this extension.

Proposition 4. *For any dl-program $(\mathcal{T}, \mathcal{P})$ such that $\mathcal{P}$ is ground, the strong answer sets of $(\mathcal{T}, \mathcal{P})$ are precisely the Herbrand interpretations of signature $\langle C, P_\mathcal{P}\rangle$ that satisfy $\mathrm{SM}^s[FO(\mathcal{P});\ P_\mathcal{P}]$ relative to $\mathcal{T}$.*

The QEL based approach was extended to cover dl-programs in [10]. In that paper, the authors capture the weak (strong, respectively) semantics of dl-programs by defining weak (strong, respectively) QHT models of dl-atoms. The two variants of F^* above are syntactic counterparts of these definitions of QHT models.

[3] The extension is similar to the extension of first-order formulas to allow aggregate expressions as given in [9].

[4] We require F to be ground because strong answer set semantics distinguishes if a ground dl-atom is monotonic or nonmonotonic.

5 Conclusion

Since the first-order stable model semantics is a generalization of the traditional stable model semantics [11] to first-order formulas, it enables a rather simple and straightforward integration of logic programs and first-order logic KB. Recent work on the first-order stable model semantics helps us in studying the semantic properties and computational aspects of the hybrid KBs. For example, as discussed, the concept of semi-safety in the first-order stable model semantics coincides with the concept of weak safety in $\mathcal{DL} + log$ and the results on semi-safety can be used to show that weak safety is a sufficient condition for ensuring the decidability of reasoning with $\mathcal{DL} + log$. Also, as discussed in [5], the notion of strong equivalence can be applied to provide the notion of equivalence between hybrid KBs.

Acknowledgements. We are grateful to David Pearce for pointers to earlier work and anonymous referees for their useful comments. The authors were partially supported by the National Science Foundation under Grants IIS-0916116 and by the IARPA SCIL program.

References

1. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artificial Intelligence 172(12-13), 1495–1539 (2008)
2. Rosati, R.: On the decidability and complexity of integrating ontologies and rules. J. Web Sem. 3(1), 61–73 (2005)
3. Rosati, R.: DL+log: Tight integration of description logics and disjunctive datalog. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR), pp. 68–78 (2006)
4. Heymans, S., de Bruijn, J., Predoiu, L., Feier, C., Nieuwenborgh, D.V.: Guarded hybrid knowledge bases. TPLP 8(3), 411–429 (2008)
5. de Bruijn, J., Pearce, D., Polleres, A., Valverde, A.: A semantical framework for hybrid knowledge bases. Knowl. Inf. Syst. 25(1), 81–104 (2010)
6. Motik, B., Rosati, R.: Reconciling description logics and rules. J. ACM 57(5) (2010)
7. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence 175, 236–263 (2011)
8. Bartholomew, M., Lee, J.: A decidable class of groundable formulas in the general theory of stable models. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR), pp. 477–485 (2010)
9. Lee, J., Meng, Y.: On reductive semantics of aggregates in answer set programming. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 182–195. Springer, Heidelberg (2009)
10. Fink, M., Pearce, D.: A logical semantics for description logic programs. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 156–168. Springer, Heidelberg (2010)
11. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of International Logic Programming Conference and Symposium, pp. 1070–1080. MIT Press, Cambridge (1988)

Gentzen-Type Refutation Systems for Three-Valued Logics with an Application to Disproving Strong Equivalence[⋆]

Johannes Oetsch and Hans Tompits

Technische Universität Wien, Institut für Informationssysteme 184/3,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,tompits}@kr.tuwien.ac.at

Abstract. While the purpose of conventional proof calculi is to axiomatise the set of valid sentences of a logic, *refutation systems* axiomatise the invalid sentences. Such systems are relevant not only for proof-theoretic reasons but also for realising deductive systems for nonmonotonic logics. We introduce Gentzen-type refutation systems for two basic three-valued logics and we discuss an application of one of these calculi for disproving strong equivalence between answer-set programs.

1 Introduction

In contrast to conventional proof calculi that axiomatise the valid sentences of a logic, *refutation systems*, also known as *complementary calculi* or *rejection systems*, are concerned with axiomatising the invalid sentences. Axiomatic rejection was introduced in modern logic by Jan Łukasiewicz in his well-known treatise on analysing Aristotle's syllogistic [1]. Subsequently, refutation systems have been studied for different logics [2,3,4,5,6,7,8] (for an overview, cf., e.g., the papers by Wybraniec-Skardowska [9] and by Caferra and Peltier [10]). Such systems are relevant not only for proof-theoretic reasons but also for realising deductive systems for nonmonotonic logics [11]. Moreover, axiomatic refutation provide the means for proof-theoretic investigations concerned with proof complexity, i.e., with the size of proof representations [12].

In this paper, we introduce analytic Gentzen-type refutation systems for two particular three-valued logics, $\mathcal{L}$ and $\mathcal{P}$, following Avron [13]. The notable feature of these logics is that they are *truth-functionally complete*, i.e., any truth-functional three-valued logic can be embedded into these logics. In particular, Gödel's three-valued logic [14] is expressible in $\mathcal{L}$, and since equivalence in this logic amounts to strong equivalence between logic programs under the answer-set semantics, in view of the well-known result by Lifschitz, Pearce, and Valverde [15], we can apply our refutation system for $\mathcal{L}$ to disprove strong equivalence between programs in a purely deductive manner, which will be briefly discussed in this paper as well. Finally, there is a Prolog implementation of our calculi available, which can be downloaded at

`www.kr.tuwien.ac.at/research/projects/mmdasp`

⋆ This work was partially supported by the Austrian Science Fund (FWF) under grant P21698. The authors would like to thank Valentin Goranko, Robert Sochacki, and Urszula Wybraniec-Skardowska for valuable support during the preparation of this paper.

2 Preliminaries

Unlike classical two-valued logic, three-valued logics admit a further truth value besides true and false. Let $\mathbf{t}$ and $\mathbf{f}$ be the classical truth values, representing true and false propositions, respectively, and $\mathbf{i}$ the third one. Semantically, there are only two major classes of three-valued logics: those where $\mathbf{i}$ is *designated*, i.e., associated with truth, and those where $\mathbf{i}$ is not designated. In this paper, we are concerned with two logics, $\mathcal{L}$ and $\mathcal{P}$ [13]. Logic $\mathcal{L}$ can be considered as a prototypical logic where $\mathbf{i}$ is not designated, whilst $\mathcal{P}$ is a prototypical logic where $\mathbf{i}$ is designated. Both logics are fully expressive, meaning that they allow to embed any truth-functional three-valued logic from the literature in it.

Both $\mathcal{L}$ and $\mathcal{P}$ are formulated over a countably infinite universe $\mathcal{U}$ of atoms including the truth constants T, F, and I. Based on the connectives $\neg$, $\vee$, $\wedge$, and $\supset$, the set of well-formed formulae is defined as usual. A set of literals is *consistent* if it does not contain both an atom and its negation. In $\mathcal{P}$, $\mathbf{t}$ and $\mathbf{i}$ are designated, while in $\mathcal{L}$, the only designated truth value is $\mathbf{t}$.

By an *interpretation*, we understand a mapping from $\mathcal{U}$ into $\{\mathbf{t}, \mathbf{f}, \mathbf{i}\}$. For any interpretation I, $I(\mathrm{T}) = \mathbf{t}$, $I(\mathrm{F}) = \mathbf{f}$, and $I(\mathrm{I}) = \mathbf{i}$. As usual, a *valuation* is a mapping from formulae into the set of truth values. We assume the ordering $\mathbf{f} < \mathbf{i} < \mathbf{t}$ on the truth values in what follows. The valuation $v^I_{\mathcal{L}}(\cdot)$ of a formula in $\mathcal{L}$ given an interpretation I is is inductively defined as follows: (i) $v^I_{\mathcal{L}}(\psi) = I(\psi)$, if ψ is an atomic formula; (ii) $v^I_{\mathcal{L}}(\neg\psi) = \mathbf{t}$ if $v^I_{\mathcal{L}}(\psi) = \mathbf{f}$, $v^I_{\mathcal{L}}(\neg\psi) = \mathbf{f}$ if $v^I_{\mathcal{L}}(\psi) = \mathbf{t}$, and $v^I_{\mathcal{L}}(\neg\psi) = \mathbf{i}$ otherwise; (iii) $v^I_{\mathcal{L}}(\psi \wedge \varphi) = \min(v^I_{\mathcal{L}}(\psi), v^I_{\mathcal{L}}(\varphi))$; (iv) $v^I_{\mathcal{L}}(\psi \vee \varphi) = \max(v^I_{\mathcal{L}}(\psi), v^I_{\mathcal{L}}(\varphi))$; and (v) $v^I_{\mathcal{L}}(\psi \supset \varphi) = v^I_{\mathcal{L}}(\varphi)$ if $v^I_{\mathcal{L}}(\psi) = \mathbf{t}$, and $v^I_{\mathcal{L}}(\psi \supset \varphi) = \mathbf{t}$ otherwise. The valuation $v^I_{\mathcal{P}}(\cdot)$ of a formula in $\mathcal{P}$ given an interpretation I is defined like $v^I_{\mathcal{L}}(\cdot)$ except for the condition of the implication: $v^I_{\mathcal{P}}(\psi \supset \varphi) = v^I_{\mathcal{P}}(\varphi)$ if $v^I_{\mathcal{P}}(\psi) = \mathbf{t}$ or $v^I_{\mathcal{P}}(\psi) = \mathbf{i}$, and $v^I_{\mathcal{P}}(\psi \supset \varphi) = \mathbf{t}$ otherwise.

A formula ψ is *true* under an interpretation I in $\mathcal{L}$ if $v^I_{\mathcal{L}}(\psi) = \mathbf{t}$. Likewise, ψ is *true* for I in $\mathcal{P}$ if $v^I_{\mathcal{P}}(\psi) = \mathbf{t}$ or $v^I_{\mathcal{P}}(\psi) = \mathbf{i}$. If ψ is true under I in $\mathcal{L}$ (resp., $\mathcal{P}$), I is a *model* of ψ in $\mathcal{L}$ (resp., $\mathcal{P}$). For a set Γ of formulae, I is a model of Γ in $\mathcal{L}$ (resp., $\mathcal{P}$) if I is a model in $\mathcal{L}$ (resp., $\mathcal{P}$) for each formula in Γ. A formula is *valid* in $\mathcal{L}$ (resp., $\mathcal{P}$) if it is true for each interpretation in $\mathcal{L}$ (resp., $\mathcal{P}$).

3 The Refutation Calculi SRCL and SRCP

Bryll and Maduch [16] axiomatised the invalid sentences of Łukasiewicz's many-valued logics including the three-valued case by means of a Hilbert-type calculus. Since their calculus is not analytic, its usefulness for proof search in practice is rather limited. In this paper, we aim at *analytic Gentzen-style refutation calculi* for three-valued logics. The first sequential refutation systems for classical propositional logic was introduced by Tiomkin [4]; equivalent systems were independently discussed by Goranko [6] and Bonatti [5]. We pursue this work towards similar refutation systems for the logics $\mathcal{L}$ and $\mathcal{P}$, which we will call **SRCL** and **SRCP**, respectively.

By an *anti-sequent*, we understand a pair of form $\Gamma \dashv \Delta$, where Γ and Δ are finite sets of formulae[1]. Given a set Γ of formulas and a formula ψ, following custom, we

[1] The symbol "$\dashv$", the dual of Frege's assertion sign "$\vdash$", is due to Ivo Thomas.

$$\frac{\Gamma \dashv \Delta, \psi}{\Gamma, \psi \supset \varphi \dashv \Delta}\ (\supset l)_1 \qquad \frac{\Gamma, \varphi \dashv \Delta}{\Gamma, \psi \supset \varphi \dashv \Delta}\ (\supset l)_2 \qquad \frac{\Gamma, \psi \dashv \Delta, \varphi}{\Gamma \dashv \Delta, \psi \supset \varphi}\ (\supset r)$$

$$\frac{\Gamma, \psi, \varphi \dashv \Delta}{\Gamma, \psi \wedge \varphi \dashv \Delta}\ (\wedge l) \qquad \frac{\Gamma \dashv \Delta, \psi}{\Gamma \dashv \Delta, \psi \wedge \varphi}\ (\wedge r)_1 \qquad \frac{\Gamma \dashv \Delta, \varphi}{\Gamma \dashv \Delta, \psi \wedge \varphi}\ (\wedge r)_2$$

$$\frac{\Gamma, \psi \dashv \Delta}{\Gamma, \psi \vee \varphi \dashv \Delta}\ (\vee l)_1 \qquad \frac{\Gamma, \varphi \dashv \Delta}{\Gamma, \psi \vee \varphi \dashv \Delta}\ (\vee l)_2 \qquad \frac{\Gamma \dashv \Delta, \psi, \varphi}{\Gamma \dashv \Delta, \psi \vee \varphi}\ (\vee r)$$

Fig. 1. Standard rules of **SRCL** and **SRCP**

$$\frac{\Gamma, \psi \dashv \Delta}{\Gamma, \neg\neg\psi \dashv \Delta}\ (\neg\neg\, l) \qquad \frac{\Gamma \dashv \Delta, \psi}{\Gamma \dashv \Delta, \neg\neg\psi}\ (\neg\neg\, r)$$

$$\frac{\Gamma, \neg\psi \dashv \Delta}{\Gamma, \neg(\psi \wedge \varphi) \dashv \Delta}\ (\neg\wedge\, l)_1 \qquad \frac{\Gamma, \neg\varphi \dashv \Delta}{\Gamma, \neg(\psi \wedge \varphi) \dashv \Delta}\ (\neg\wedge\, l)_2$$

$$\frac{\Gamma \dashv \Delta, \neg\psi, \neg\varphi}{\Gamma \dashv \Delta, \neg(\psi \wedge \varphi)}\ (\neg\wedge\, r) \qquad \frac{\Gamma, \neg\psi, \neg\varphi \dashv \Delta}{\Gamma, \neg(\psi \vee \varphi) \dashv \Delta}\ (\neg\vee\, l)$$

$$\frac{\Gamma \dashv \Delta, \neg\psi}{\Gamma \dashv \Delta, \neg(\psi \vee \varphi)}\ (\neg\vee\, r)_1 \qquad \frac{\Gamma \dashv \Delta, \neg\varphi}{\Gamma \dashv \Delta, \neg(\psi \vee \varphi)}\ (\neg\vee\, r)_2$$

$$\frac{\Gamma, \psi, \neg\varphi \dashv \Delta}{\Gamma, \neg(\psi \supset \varphi) \dashv \Delta}\ (\neg\supset l) \qquad \frac{\Gamma \dashv \Delta, \psi}{\Gamma \dashv \Delta, \neg(\psi \supset \varphi)}\ (\neg\supset r)_1$$

$$\frac{\Gamma \dashv \Delta, \neg\varphi}{\Gamma \dashv \Delta, \neg(\psi \supset \varphi)}\ (\neg\supset r)_2$$

Fig. 2. Non-Standard rules of **SRCL** and **SRCP**

write "Γ, ψ" as a shorthand for $\Gamma \cup \{\psi\}$. An interpretation I *refutes* $\Gamma \dashv \Delta$ in $\mathcal{L}$ (resp., $\mathcal{P}$) iff I is a model of Γ in $\mathcal{L}$ (resp., $\mathcal{P}$) and all formulae in Δ are false under I in $\mathcal{L}$ (resp., $\mathcal{P}$). Moreover, an anti-sequent is *refutable* in $\mathcal{L}$ (resp., $\mathcal{P}$) iff it is refuted by some interpretation in $\mathcal{L}$ (resp., $\mathcal{P}$).

The postulates of the calculi **SRCL** and **SRCP** are as follows: Let Γ and Δ be two disjoint sets of literals such that $\neg\mathrm{T}, \mathrm{F} \notin \Gamma$ and $\mathrm{T}, \neg\mathrm{F} \notin \Delta$. Then, $\Gamma \dashv \Delta$ is an axiom of **SRCL** iff $\{\mathrm{I}, \neg\mathrm{I}\} \cap \Gamma = \emptyset$ and Γ is consistent, and $\Gamma \dashv \Delta$ is an axiom of **SRCP** iff $\{\mathrm{I}, \neg\mathrm{I}\} \cap \Delta = \emptyset$ and Δ is consistent. The inference rules of **SRCL** and **SRCP** comprise the *standard rules* depicted in Fig. 1 and the *non-standard rules* depicted in Fig. 2. The standard rules introduce one occurrence of $\wedge$, $\vee$, or $\supset$ at a time. Note that they coincide with the respective introduction rules in the refutation systems for classical logic [4,6,5]. The non-standard rules introduce two occurrences of a connective at the same time, in particular this concerns negation in combination with all other connectives. Note that the logical rules of **SRCL** and **SRCP** coincide, so the difference between the two calculi lies only in their axioms.

Theorem 1 (Soundness and Completeness). *For any anti-sequent* $\Gamma \dashv \Delta$*, (i)* $\Gamma \dashv \Delta$ *is provable in* $\mathbf{SRCL}$ *iff* $\Gamma \dashv \Delta$ *is refutable in* $\mathcal{L}$*, and (ii)* $\Gamma \dashv \Delta$ *is provable in* $\mathbf{SRCP}$ *iff* $\Gamma \dashv \Delta$ *is refutable in* $\mathcal{P}$*.*

Note that our calculi are, in a sense, refutational counterparts of the Gentzen-type calculi of Avron [17] for axiomatising the valid sentences of $\mathcal{L}$ and $\mathcal{P}$. In fact, for each unary rule in Avron's systems, our system contains a respective rule were "$\vdash$" is replaced by "$\dashv$", whilst for each binary rule of form

$$\frac{\Gamma' \vdash \Delta' \quad \Gamma'' \vdash \Delta''}{\Gamma \vdash \Delta}$$

of Avron, our systems contain two rules

$$\frac{\Gamma' \dashv \Delta'}{\Gamma \dashv \Delta} \quad \text{and} \quad \frac{\Gamma'' \dashv \Delta'}{\Gamma \dashv \Delta}.$$

Hence, as already remarked by Bonatti [5], exhaustive search in the standard system becomes non-determinism in the refutation system—a property that often allows for quite concise proofs and thus helps to reduce the size of proof representations.

Contrary to standard sequential systems, our systems do not contain binary rules. Hence, proofs in our systems are not trees but sequences, and consequently each proof has a single axiom. In fact, a proof of a formula ψ does not represent a single counter model for ψ, rather it represents an entire class of counter models for ψ, in view of the following property underlying the soundness of our calculi: each interpretation I that refutes the axiom $\Gamma \dashv \varDelta$ in a proof of $\dashv \psi$ in $\mathbf{SRCL}$ (resp., in $\mathbf{SRCP}$), also refutes $\dashv \psi$ in $\mathbf{SRCL}$ (resp., in $\mathbf{SRCP}$).

4 An Application for Disproving Strong Equivalence

We outline an application scenario that is concerned with logic programs under the answer-set semantics [18]. In a nutshell, a (*disjunctive*) *logic program* is a set of rules of form $a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \text{not } a_{m+1}, \ldots \text{not } a_n$, where all a_i are atoms over some universe $\mathcal{U}$ and "not" denotes default negation. The answer-sets of a program are sets of atoms defined using a fixed-point construction based on the reduct of a program relative to an interpretation [18].

Two logic programs are *equivalent* if they have the same answer sets. In contrast to classical logic, equivalence between programs fails to yield a replacement property. The notion of *strong equivalence* circumvents this problem: two programs P and Q are strongly equivalent iff, for each program R, $P \cup R$ and $Q \cup R$ are equivalent. For instance, consider $P = \{a \leftarrow \text{not } b,\ b \leftarrow \text{not } a\}$ and $Q = \{a \vee b\}$. P and Q are equivalent but not strongly equivalent.

The central observation connecting strong equivalence with three-valued logics is the well-known result [15] that strong equivalence between two programs P and Q holds iff P and Q, interpreted as theories, are equivalent in Gödel's three-valued logic [14]. The connectives of three-valued Gödel logic are $\wedge$, $\vee$, $\sim$, and $\rightarrow_G$, which can be defined in $\mathcal{L}$ as $\sim\psi = \neg(\neg\psi \supset \psi)$ and $\psi \rightarrow_G \varphi = ((\neg\varphi \supset \neg\psi) \supset \psi) \supset \varphi$. In view of this, we can extend $\mathbf{SRCL}$ by derived rules for $\sim$ and $\rightarrow_G$, which are given in Fig. 3.

$$\dfrac{\Gamma, \neg\psi \dashv \Delta}{\Gamma, {\sim}\psi \dashv \Delta}\ ({\sim}\, l) \qquad \dfrac{\Gamma \dashv \Delta, \neg\psi}{\Gamma \dashv \Delta, {\sim}\psi}\ ({\sim}\, r)$$

$$\dfrac{\Gamma \dashv \Delta, \psi, \neg\varphi}{\Gamma, \psi \to_G \varphi \dashv \Delta}\ (\to_G\ l)_1 \qquad \dfrac{\Gamma, \varphi \dashv \Delta}{\Gamma, \psi \to_G \varphi \dashv \Delta}\ (\to_G\ l)_2$$

$$\dfrac{\Gamma, \neg\psi \dashv \Delta}{\Gamma, \psi \to_G \varphi \dashv \Delta}\ (\to_G\ l)_3 \qquad \dfrac{\Gamma, \psi \dashv \Delta, \varphi}{\Gamma \dashv \Delta, \psi \to_G \varphi}\ (\to_G\ r)_1$$

$$\dfrac{\Gamma, \neg\varphi \dashv \Delta, \neg\psi}{\Gamma \dashv \Delta, \psi \to_G \varphi}\ (\to_G\ r)_2 \qquad \dfrac{\Gamma \dashv \Delta, \neg\psi}{\Gamma, \neg{\sim}\psi \dashv \Delta}\ (\neg{\sim}\, l)$$

$$\dfrac{\Gamma, \neg\psi \dashv \Delta}{\Gamma \dashv \Delta, \neg{\sim}\psi}\ (\neg{\sim}\, r) \qquad \dfrac{\Gamma, \neg\varphi \dashv \Delta, \neg\psi}{\Gamma, \neg(\psi \to_G \varphi) \dashv \Delta}\ (\neg{\to_G}\ l)$$

$$\dfrac{\Gamma \dashv \Delta, \neg\varphi}{\Gamma \dashv \Delta, \neg(\psi \to_G \varphi)}\ (\neg{\to_G}\ r)_1 \qquad \dfrac{\Gamma, \neg\psi \dashv \Delta}{\Gamma \dashv \Delta, \neg(\psi \to_G \varphi)}\ (\neg{\to_G}\ r)_2$$

Fig. 3. Derived rules for three-valued Gödel logic

To verify that P and Q are not strongly equivalent, it suffices to give a proof of one of $P \dashv Q$ or $Q \dashv P$ in **SRCL**[2]. While $Q \dashv P$ is not provable, there is a proof of $P \dashv Q$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dashv a, b, \neg a, \neg b}{\dashv a, b, {\sim}a, {\sim}b}\ ({\sim}\, r), ({\sim}\, r)}{{\sim}a \to_G b, {\sim}b \to_G a \dashv a, b}\ (\to_G\ l)_1, (\to_G\ l)_1}{{\sim}a \to_G b, {\sim}b \to_G a \dashv a \vee b}\ (\vee\, r)}{({\sim}a \to_G b) \wedge ({\sim}b \to_G a) \dashv a \vee b}\ (\wedge\, l)$$

Hence, P and Q are indeed not strongly equivalent. In fact, as detailed below, a concrete program R such that $P \cup R$ and $Q \cup R$ have different answer sets, i.e., a *witness* that P and Q are not strongly equivalent, can be immediately constructed from the axiom $\dashv a, b, \neg a, \neg b$ of the above proof: $R = \{a \leftarrow b,\ b \leftarrow a\}$. Indeed, $P \cup R$ has no answer set while $Q \cup R$ yields $\{a, b\}$ as its unique answer set.

The general method to obtain a witness theory (as R above) from an axiom in **SRCL** is as follows: Given an axiom $\Gamma \dashv \Delta$, construct some interpretation I that refutes $\Gamma \dashv \Delta$. For the above example, an interpretation that assigns both a and b to $\mathbf{i}$ would refute the axiom already. Note that I then refutes $P \dashv Q$ as well. Based on I, a witness program R can always be constructed by using the next proposition which immediately follows from the proof of the main theorem by Lifschitz, Pearce, and Valverde [15]:

Proposition 1. *Let P and Q be two programs such that an I is a model of P but not of Q in three-valued Gödel logic, and let J be the classical interpretation defined by setting $J(a) = \mathbf{f}$ iff $I(a) = \mathbf{f}$, and define $R' = \{a \mid I(a) = \mathbf{t}$ or $I(a) = \mathbf{i}\}$ and*

[2] We interpret programs as a theories, i.e., as the conjunctions of rules, where rules are interpreted as implications.

$R'' = \{a \mid I(a) = \mathbf{t}\} \cup \{a \rightarrow_G b \mid I(a) = I(b) = \mathbf{i}\}$. *Then,* $P \cup R$ *and* $Q \cup R$ *are not strongly equivalent, where* $R = R'$ *if* J *is not a classical model of* Q*, and* $R = R''$ *otherwise.*

Note that a proof that two programs are not strongly equivalent represents, in general, not only a single witness program but an entire class of programs which distinguishes our axiomatic approach from approaches based on finding counter models.

References

1. Łukasiewicz, J.: Aristotle's syllogistic from the standpoint of modern formal logic, 2nd edn. Clarendon Press, Oxford (1957)
2. Kreisel, G., Putnam, H.: Eine Unableitbarkeitsbeweismethode für den Intuitionistischen Aussagenkalkül. Archiv für Mathematische Logik und Grundlagenforschung 3, 74–78 (1957)
3. Wójcicki, R.: Dual counterparts of consequence operations. Bulletin of the Section of Logic 2, 54–57 (1973)
4. Tiomkin, M.: Proving unprovability. In: 3rd Annual Symposium on Logics in Computer Science, pp. 22–27. IEEE, Los Alamitos (1988)
5. Bonatti, P.A.: A Gentzen system for non-theorems. Technical Report CD-TR 93/52, Christian Doppler Labor für Expertensysteme, Technische Universität Wien (1993)
6. Goranko, V.: Refutation systems in modal logic. Studia Logica 53, 299–324 (1994)
7. Skura, T.: Refutations and proofs in S4. In: Proof Theory of Modal Logic, pp. 45–51. Kluwer, Dordrecht (1996)
8. Skura, T.: A refutation theory. Logica Universalis 3, 293–302 (2009)
9. Wybraniec-Skardowska, U.: On the notion and function of the rejection of propositions. Acta Universitatis Wratislaviensis Logika 23, 179–202 (2005)
10. Caferra, R., Peltier, N.: Accepting/rejecting propositions from accepted/rejected propositions: A unifying overview. International Journal of Intelligent Systems 23, 999–1020 (2008)
11. Bonatti, P.A., Olivetti, N.: Sequent calculi for propositional nonmonotonic logics. ACM Transactions on Computational Logic 3, 226–278 (2002)
12. Egly, U., Tompits, H.: Proof-complexity results for nonmonotonic reasoning. ACM Transactions on Computational Logic 2, 340–387 (2001)
13. Avron, A.: Natural 3-valued logics - Characterization and proof theory. Journal of Symbolic Logic 56 (1), 276–294 (1991)
14. Gödel, K.: Zum intuitionistischen Aussagenkalkül. Anzeiger Akademie der Wissenschaften Wien, mathematisch-naturwissenschaftliche Klasse 32, 65–66 (1932)
15. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2, 526–541 (2001)
16. Bryll, G., Maduch, M.: Aksjomaty odrzucone dla wielowartościowych logik Łukasiewicza. In: Zeszyty Naukowe Wyższej Szkły Pedagogigicznej w Opolu, Matematyka VI, Logika i algebra, pp. 3–17 (1968)
17. Avron, A.: Classical Gentzen-type methods in propositional many-valued logics. In: 31st IEEE International Symposium on Multiple-Valued Logic, pp. 287–298. IEEE, Los Alamitos (2001)
18. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)

New Semantics for Epistemic Specifications

Michael Gelfond

Texas Tech University
michael.gelfond@ttu.edu

Abstract. This note presents a new version of the language of epistemic specifications. The semantics of the new language is arguably closer to the intuitive meaning of epistemic operators. It eliminates some unintended interpretations which exist under the old definition. The author hopes that the new language will be better suited for the design of intelligent agents capable of introspective reasoning with incomplete information.

1 Introduction

The language of epistemic specifications [1,4] is an extension of the language of disjunctive logic programs [3] by modal operators K and M, where KF stands for F *is known to be true* and MF stands for F *may be believed to be true.* The stated purpose of this work was to *allow for the correct representation of incomplete information in the presence of multiple belief sets.* To illustrate the need for the extension the authors considered the following simple example.

Example 1. [Scholarship Eligibility]
Consider a collection of rules

1. $eligible(X) \leftarrow highGPA(X)$
2. $eligible(X) \leftarrow minority(X), fairGPA(X)$
3. $\neg eligible(X) \leftarrow \neg fairGPA(X), \neg highGPA(X)$
4. $interview(X) \leftarrow \ not \ \ eligible(X),$
   $\qquad\qquad\qquad\quad not \ \ \neg eligible(X)$

used by a certain college for awarding scholarships to its students. The first three rules are self explanatory (we assume that variable X ranges over a given set of students) while the fourth rule can be viewed as a formalization of the statement:

(*) *The students whose eligibility is not determined by the college rules should be interviewed by the scholarship committee.*

We assume that this program is to be used in conjunction with a database DB consisting of literals specifying values of the predicates $minority$, $highGPA$, and $fairGPA$. Consider, for instance, DB consisting of the following two facts about one of the students:

5. $fairGPA(ann)$
6. $\neg highGPA(ann)$

(Notice that DB contains no information about the minority status of Ann.) It is easy to see that rules (1)–(6) allow us to conclude neither $eligible(ann)$ nor $\neg eligible(ann)$, therefore eligibility of Ann for the scholarship is undetermined and, by rule (4), she must be interviewed. Formally this argument is reflected by the fact that program T_1 consisting of rules (1)–(6) has exactly one answer set:

$$\{fairGPA(ann), \neg highGPA(ann), interview(ann)\}.$$

The situation changes significantly if disjunctive information about students is allowed to be represented in the database. Suppose, for instance, that we need to augment rules (1)–(3) by the following information:

(**) Mike's GPA is fair or high.

The corresponding program T_2 consists of rules (1)–(3) augmented by the disjunction

7. $fairGPA(mike)$ *or* $highGPA(mike)$

T_2 has two answer sets:

$$A_1 = \{highGPA(mike), eligible(mike)\}$$

and

$$A_2 = \{fairGPA(mike)\},$$

and therefore the reasoner associated with T_2 does not have enough information to establish Mike's eligibility for the scholarship (i.e. his answer to query ? $eligible(mike)$ will be *unknown*). If we now expand this theory by (*) we expect the new theory T_3 to be able to answer *yes* to a query $interview(mike)$. It is easy to see however that if (*) is represented by (4) this goal is not achieved. The resulting theory T_3 consisting of (1)–(4) and (7) has two answer sets

$$A_3 = \{highGPA(mike), eligible(mike)\}$$

$$A_4 = \{fairGPA(mike), interview(mike)\}$$

and therefore the answer to query $interview(mike)$ is *unknown*. The reason of course is that (4) is too weak to represent (*). The informal argument we are trying to capture goes something like this: theory T_3 answers neither *yes* nor *no* to the query $eligible(mike)$. Therefore, the answer to this question is undetermined, and, by (*), Mike should be interviewed. To formalize this argument our system should have a more powerful introspective ability than the one captured by the notion of answer sets from [1].

To remedy this problem [1,4] introduced the language of epistemic specifications. Literals of this language were divided into

– *objective* – expressions of the form $p(\bar{t})$, $\neg p(\bar{t})$, and
– *subjective* – expressions of the form $\mathrm{K}l$, $\neg\mathrm{K}l$, $\mathrm{M}l$, $\neg\mathrm{M}l$ where l is an objective literal.

Epistemic specifications were defined as collections of rules of the form:

$$l_1 \ or \ \ldots \ or \ l_k \leftarrow g_{k+1}, \ldots, g_m, not \ \ l_{m+1}, \ldots, not \ \ l_n \tag{1}$$

where the l's are objective literals and the g's are subjective or objective literals[1]. The semantics of an epistemic specification T has been given via the notion of a *world view* of T - a collection of simple theories about the world which can be built by a rational reasoner on the instructions from T. For a program T not containing operators K and M the world view of T coincided with the collection of all the answer sets of T. The precise definition went as follows:

Let T be a ground epistemic specifications and S be a collection of sets of ground objective literals in the language of T; S entails K l ($S \models$ K l) if for every $W \in S$, $l \in W$. Otherwise $S \models \neg$K l). Similarly for M.
A disjunctive logic program T^S was obtained from T by:

1. removing from T all rules containing subjective literals not entailed by S.
2. removing from rules in T all other occurrences of subjective literals.

T^S was referred to as the reduct of T with respect to S. A set S was called a *world view* of T if S were the collection of all answer sets of T^S. Elements of S were called *belief sets* of T. Epistemic specifications with variables were viewed as shorthands for the collection of their ground instances.

The following example shows that the language of epistemic specifications provided the way to deal with the problem outlined in example 1.

Example 2. [Example 1 revisited]
The statement (*) above could be naturally expressed in the language of epistemic specifications by the rule:

$$\begin{aligned} interview(X) \leftarrow & \ not \ \ \mathrm{K}eligible(X), \\ & \ not \ \ \mathrm{K}\neg eligible(X) \end{aligned}$$

which corresponds closely to the intuitive meaning of (*). The epistemic specification T consisting of this rule together with the rules (1) – (3) and (7) from Example 1 has the world view $A = \{A_1, A_2\}$ where

$$A_1 = \{highGPA(mike), eligible(mike), interview(mike)\}$$

$$A_2 = \{fairGPA(mike), interview(mike)\}$$

Therefore T answers *unknown* to the query $eligible(mike)$ and *yes* to the query $interview(mike)$ which is the intended behavior of the system.

Unfortunately, as was first noticed by Teodor Przimusinski, world views of epistemic specifications do not always correspond to those intended by the authors. Consider for instance the following example:

[1] The actual language defined in these papers is substantially more general but its simple version we present here is sufficient for our purpose.

Example 3. [Unsupported Beliefs]
Consider epistemic specification T_1 consisting of the rule

$p \leftarrow \mathrm{K}p$

It is easy to check that it has two world views: $A_1 = \{\emptyset\}$ and $A_2 = \{\{p\}\}$. Clearly the second one is unintended. A rational agent will not have a belief p which is supported only by $\mathrm{K}p$.

Even though some attempts to remedy the situation were made in [2] we have never been able to obtain a fully satisfactory solution to the problem of unintended world views. This paper is another attempt to the solution. The work is of course preliminary but the author thought that it may be worth publishing since there seems to be some renewed interest in epistemic specifications (see, for instance, [6],[7].

2 The New Definition of Epistemic Specifications

The new definition of epistemic specifications suggests changes in both, syntax and semantics of the language. Objective literals are defined as before; the subjective literals have the form $\mathrm{K}\ l$ where l is an objective literal possibly preceded by the default negation *no*. Expression K *not* p, which was not allowed in the old version becomes a subjective literal of the new language. According to the new definition *epistemic specification* is a collection of rules of the form:

$$l_1\ or\ \ldots\ or\ l_k \leftarrow g_{k+1}, \ldots, g_m, not\ \ l_{m+1}, \ldots, not\ \ l_n \tag{2}$$

where the l's are objective literals and the g's are subjective or objective literals. The new syntax allows modal operator M to be expressed in terms of K.

$$\mathrm{M}\, l =_{def} \neg K\ not\ \ l$$

As before, programs with variables are viewed as shorthands for their ground instantiations. The second, more substantial, change is in the definition of the notion of the reduct.

Definition 1. *[New Reduct]*
Let T be an epistemic specification and S be a collection of sets of ground literals in the language of T. By T^S we will denote the disjunctive logic program obtained from T by:

1. removing all rules containing subjective literals g such that $S \not\models g$,
2. removing all other occurrences of subjective literals of the form $\neg\mathrm{K}\ l$,
3. replacing remaining occurrences of literals of the form $\mathrm{K}\ l$ by l.

The definition of world view of T remains unchanged: a set S is called a *world view* of T if S is the collection of all answer sets of T^S.

The new definition deals correctly with the eligibility examples. According to the new definition specification T from Example 2 has exactly the world views it had under the old definition. This is not surprising, since the rules of T contain no positive occurrences of K and hence the old reduct coincides with the new one.

Let us now see how the new definition deals with unintended world views.

Example 4. [Example 3 revisited]
Consider

$T_1 = \{p \leftarrow Kp\}$

from Example 3. It is easy to see that, under the new semantics, $A_1 = \{\emptyset\}$ remains a world view of T_1. However, $A_2 = \{\{p\}\}$ is not a world view of T_1 according to the new definition; $T_1^{A_2} = \{p \leftarrow p\}$. Clearly its answer set, $\emptyset$, is not equal to $\{p\}$. The new definition of reduct helps to eliminate the unsupported beliefs.

The next example shows how the new language can be used for an alternative formalization of the *closed world assumption* (CWA) [5]. The assumption, which says that $p(X)$ should be assumed to be false if there is no evidence to the contrary, is normally expressed by an ASP rule

$$\neg p(X) \leftarrow \textit{not } p(X) \tag{3}$$

An interesting alternative representation of CWA may be given by epistemic rule

$$\neg p(X) \leftarrow \neg \mathrm{M} p(X) \tag{4}$$

To better understand this formalization let us consider

Example 5. [Closed World Assumption]
Let T consist of the rules

1. $p(a)$ *or* $p(b)$.
2. $p(c)$.
3. $q(d)$.
4. $\neg p(X) \leftarrow \neg \mathrm{M} p(X)$

According to the new definition the specification has one world view,

$$A = \{\{q(d), p(a), p(c), \neg p(d)\},\ \{q(d), p(b), p(c), \neg p(d)\}\}.$$

To see that it is enough to recall that the rule

$$\neg p(X) \leftarrow \neg \mathrm{M} p(X)$$

is a shorthand for

$$\neg p(X) \leftarrow \mathrm{K}\ \textit{not } p(X)$$

and hence the corresponding reduct is

1. $p(a)$ *or* $p(b)$.
2. $p(c)$.
3. $q(d)$.
4. $\neg p(d) \leftarrow not\ \ p(d)$

If, however, we replace the last rule by the rule 3 the world view will change. Now it would be

$$B = \{\{q(d), p(a), \neg p(b), p(c), \neg p(d)\},\ \{q(d), p(b), \neg p(a), p(c), \neg p(d)\}\}.$$

Hence the first program will answer *unknown* to a query $\neg p(a)$ *or* $\neg p(b)$ while the second will answer *yes*. To have the semantics of M in which program T above will have exactly one world view, A, was part of the original goal of [1,4]. Unfortunately, however, the goal was not achieved. According to the old definition specification T above has three world views: the world view A above and

$A_1 = \{\{q(d),\ p(c),\ p(a),\ \neg p(b),\ \neg p(d)\}\}$,

$A_2 = \{\{q(d),\ p(c),\ p(b),\ \neg p(a),\ \neg p(d)\}\}$,

The new definition of reduct allows us to get rid of the two unintended world views.

Obviously the work presented in this note is preliminary. The first next step is to see if the known results establishing properties of epistemic specifications and the corresponding reasoning algorithms can be adopted to the new language. There are many possible applications. Most immediate ones are to investigate the use of the language for conformant planning and, when suitably expanded, for probabilistic reasoning.

References

1. Gelfond, M.: Epistemic approach to formalization of commonsense reasoning. Technical Report TR-91-2, University of Texas at El Paso (1991)
2. Gelfond, M.: Logic programming and reasoning with incomplete information. Annals of Mathematics and Artificial Intelligence 12 (1994)
3. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3/4), 365–386 (1991)
4. Gelfond, M., Przymusinska, H.: Reasoning in open domains. In: In Logic Programming and Non-Monotonic Reasoning, pp. 397–413. MIT Press, Cambridge (1993)
5. Reiter, R.: On Closed World Data Bases. In: Logic and Data Bases, pp. 119–140. Plenum Press, New York (1978)
6. Truszczynski, M.: Revisiting epistemic specifications. In: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond (2010)
7. Zhang, Y.: Updating epistemic logic programs. J. Logic Computation 19(2), 405–423 (2009)

cmMUS: A Tool for Circumscription-Based MUS Membership Testing

Mikoláš Janota[2] and Joao Marques-Silva[1,2]

[1] University College Dublin, Ireland
[2] INESC-ID, Lisbon, Portugal

Abstract. This article presents cmMUS—a tool for deciding whether a clause belongs to some minimal unsatisfiable subset (MUS) of a given formula. While MUS-membership has a number of practical applications, related with understanding the causes of unsatisfiability, it is computationally challenging—it is Σ_2^P-complete. The presented tool cmMUS solves the problem by translating it to propositional circumscription, a well-known problem from the area of non-monotonic reasoning. The tool constantly outperforms other approaches to the problem, which is demonstrated on a variety of benchmarks.

1 Introduction

Unsatisfiable formulas, representing refutation-proofs or inconsistencies, appear in various areas of automated reasoning. This article presents a tool that helps us to understand why a certain formula is unsatisfiable. To understand why a formula in the conjunctive normal form (CNF), is unsatisfiable, it is sufficient to consider only some of its subsets of clauses. More precisely, a set of clauses is called a minimally unsatisfiable subset (MUS) if it is unsatisfiable and any of its subsets is satisfiable. cmMUS determines whether a given clause belongs to *some* MUS. This is referred to as the MUS-MEMBERSHIP problem.

The MUS-MEMBERSHIP problem is important when one wants to *restore consistency* of a formula: removing a clause that is *not* part of any MUS, will certainly not restore consistency. Restoring consistency is an active area of research in the area of *product configuration* [16,17]. For example, when configuring a product, some sets of features result in an inconsistent configuration. Approaches for resolving conflicting features often involves user intervention, e.g. to decide which features to remove. Clearly, it is preferable to allow the user to deselect features relevant for the inconsistency.

2 Background

Throughout this paper, ϕ and ψ denote Boolean formulas. A Boolean formula ϕ in Conjunctive Normal Form (CNF) is a conjunction of disjunctions of literals. Each disjunction of literals is called a *clause*, and it is preferably represented by ω. Where appropriate, a CNF formula is interpreted as a set of clauses. A *truth assignment* μ_X is a mapping from a set of variables X to $\{0, 1\}$, $\mu_X : X \to \{0, 1\}$.

A *QBF formula* is a Boolean formula where variables can be universally or existentially quantified. We write QBF$_{k,\exists}$ to denote the class of formulas of the form $\exists X_1 \forall Y_1 \ldots \exists X_k \forall Y_k.\ \phi$. An important result from the complexity theory is that the validity of a formula in QBF$_{k,\exists}$ is Σ_k^P-complete [15].

A *Disjunctive Logic Program* (*DLP*) is a set of rules of the form $a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_m, {\sim}c_1, \ldots, {\sim}c_k$, where a_j's, b_i's, c_l's are propositional atoms. The part $a_1 \vee \cdots \vee a_n$ is called the *head* and is viewed as a disjunction. The part right of $\leftarrow$ is called the *body* and is viewed as a conjunction. The symbol $\sim$ is the *default negation* (the failure to prove). The empty head is denoted as $\bot$. If a program comprises only rules with $k = 0$, the program is called *positive*. The *stable model semantics* is assumed for disjunctive logic programs [4,5].

Circumscription was introduced by McCarthy as a form of nonmonotonic reasoning [14]. While the original definition of circumscription is for first-order logic, for the purpose of this article we consider its propositional version. For a set of variables R, a model M of the formula ϕ is an *R-minimal model* if it is minimal with respect to the point-wise ordering on the variables R. The *circumscription inference problem* is the problem of deciding whether a formula ψ holds in all R-minimal models of a formula ϕ. If a formula ψ holds in all R-minimal models of a formula ϕ we write $\phi \models_R^{circ} \psi$.

We say that a set of clauses $\psi \subseteq \phi$ is a *Maximally Satisfiable Subformula (MSS)* iff ψ is satisfiable and any set ψ' s.t. $\psi \subsetneq \psi'$ is unsatisfiable. Dually, we say that a set of clauses $\psi \subseteq \phi$ is a *Minimally Unsatisfiable Subformula (MUS)* iff ψ is unsatisfiable and any set $\psi' \subsetneq \psi$ is satisfiable. The definition of MUSes yields the following problems.

Name: MUS-MEMBERSHIP
Given: A CNF formula ϕ and a clause ω.
Question: Is there an MUS ψ of ϕ such that $\omega \in \psi$?

Name: MUS-OVERLAP
Given: CNF formulas ϕ and γ.
Question: Is there an MUS ψ of ϕ such that $\gamma \cap \psi \neq \emptyset$?

Observe that MUS-MEMBERSHIP is a special case of MUS-OVERLAP when γ consists of a single clause, and, that MUS-OVERLAP can be expressed as a disjunction of k instances of MUS-MEMBERSHIP, where k is the number of clauses in the formula γ. However, cmMUS solves directly the more general problem MUS-OVERLAP.

It has been shown that both MUS-MEMBERSHIP and entailment in propositional circumscription are in the second level of the polynomial hierarchy: MUS-MEMBERSHIP (and therefore MUS-OVERLAP) is Σ_2^P-complete [12]; entailment in propositional circumscription is Π_2^P-complete [2].

3 cmMUS Description

cmMUS[1] solves MUS-OVERLAP by translating it to propositional circumscription entailment. It accepts a formula in the DIMACS format and a list of indices representing

[1] Available at `http://sat.inesc-id.pt/~mikolas/sw/cmmus`

the clauses tested for overlap. To decide MUS-OVERLAP for a formula ϕ and a set of clauses γ the tool performs the following steps.

1. It introduces the *relaxed form* of the formula $\phi^* = \{\omega \vee r_\omega \mid \omega \in \phi\}$, where r_ω are fresh variables.
2. It generates the circumscription entailment problem $\phi^* \models_R^{circ} \bigwedge_{\omega \in \gamma} \neg r_\omega$.
3. It solves the entailment by a dedicated algorithm based on counterexample guided abstraction refinement [9].
4. If the answer to the entailment problem is "valid", then there is no overlap between MUSes of ϕ and the clauses γ. If the answer to the entailment problem is "invalid", then there is an overlap. Further, if r_ω has the value 1 in a counterexample to the entailment, the clause ω overlaps with some MUS of ϕ.

Apart from the "yes"/"no" answer to the given MUS-OVERLAP problem, in the case of an overlap ("yes"), the tool outputs a formula $\phi' \subseteq \phi$ such that ϕ' is unsatisfiable and any of its MUSes overlaps with γ. Details of the translation are explained in the pertaining technical report [10].

4 Experimental Results

The following tools were considered for the experimental evaluation in addition to cmMUS.

look4MUS is a tool dedicated to MUS-MEMBERSHIP based on MUS enumeration, guided by heuristics based on a measure of inconsistency [7].

Quantified Boolean Formula (QBF). The problem was expressed as a QBF [10] and inputted to the QBF solver `QuBE7.1`[2]. The solver was chosen because it solved the most instances in the 2QBF track of QBF Evaluation 2010[3]. The solver was invoked with all its preprocessing techniques (using the `-all` switch) [6].

MSS enum. A clause appears in some MUS if there exists an MSS that does not contain it [11,10]. The tool `CAMUS` [13] was used to enumerate MSSes of the given formula. The enumeration stops if it encounters an MSS that does not contain at least one of the clauses in γ.

Disjunctive Logic Programming. The translation to disjunctive logic programming (DLP) was performed in a sequence of steps.

1. Translate the relaxed version ϕ^* into a positive disjunctive logic program by putting positive atoms in the head and negative in the body.
2. Apply the tool `circ2dlp` [8] to produce a disjunctive logic program whose stable models correspond to the R-minimal models of the given formula.
3. To find out whether the set of clauses $\omega_1, \omega_2, ..., \omega_n$ overlaps with any MUS of ϕ^*, add to this program the rule $\perp \leftarrow \sim r_{\omega_1}, \sim r_{\omega_2}, ..., \sim r_{\omega_n}$ which disables the stable models where none of the clauses in question are relaxed. The resulting program has at least one model iff there exists an MSS such that at least one of the clauses in question is relaxed, an approach suggested in [3].
4. Run the DLP solver `claspD` [1] to decide whether it has at least one model are not.

[2] Available at `www.star.dist.unige.it/~qube/`

[3] `http://www.qbflib.org/`

Table 1. Number of solved instances by the different approaches

	cmMUS	look4MUS	MSS enum.	QBF	DLP
Nemesis (bf) (223)	223	223	31	8	0
Daimler-Chrysler (84)	46	13	49	0	0
dining philosophers (22)	18	17	4	0	0
dimacs (87)	87	82	51	48	0
ezfact (41)	21	11	11	0	0
crafted (24)	24	14	13	12	5
total (481)	419	360	159	68	5

A variety of unsatisfiable formulas was selected from the benchmarks used for SAT competitions[4] and from well-known applications of SAT (namely ATPG and product configuration). The selected formulas are relatively easy for modern SAT solvers because MUS-MEMBERSHIP is significantly harder than satisfiability. Even so, instances with tens of thousands of clauses were used (e.g. dining philosophers).

For each of these formulas the MUS-OVERLAP was computed using the various approaches. The 1st, 3rd, 5th, and 7th clauses in the formula's representation were chosen as the set γ for which the overlap was to be determined—this testing methodology was also used in [7].

All experimental results were obtained on an Intel Xeon 5160 3GHz with 4GB of memory. The experiments were obtained with a time limit of 1,000 seconds. The results of the measurements are presented by Table 1 and Figure 1. Table 1 presents the number of solved instances by each of the approaches for each set of benchmarks. Figure 1 presents the computation times with *cactus plots*—the horizontal axis represents the number of instances that were solved within the time represented by the vertical axis.

Out of the presented approaches, the cmMUS is the most robust one: it has solved the most instances (419) and except for one class of benchmarks it exhibits the shortest overall running times. The set of benchmarks where cmMUS came second are the Daimler-Chrysler. In these benchmarks the simple MSS enumeration solved 3 more instances.

The dedicated algorithm `look4MUS` came second in terms of number of the solved instances (360) and it solved a number of benchmarks in a short time (Nemesis-bf), although slower than cmMUS. However, it turned out not to be robust (e.g. a small number of instances were solved in suite Daimler-Chrysler and ezfact).

The QBF and DLP approaches turned out to be the least successful ones. In the case of DLP this is most likely attributed to the relatively small number of variables on which the circumscription is being minimized (the set P). This weakness has already been highlighted by the authors of `circ2dlp` [8]. However, to our knowledge, the solver `claspD` does not use such extensive preprocessing techniques as `Qube 7.1`. Hence, this could be investigated in the future.

[4] `http://www.satcompetition.org/`

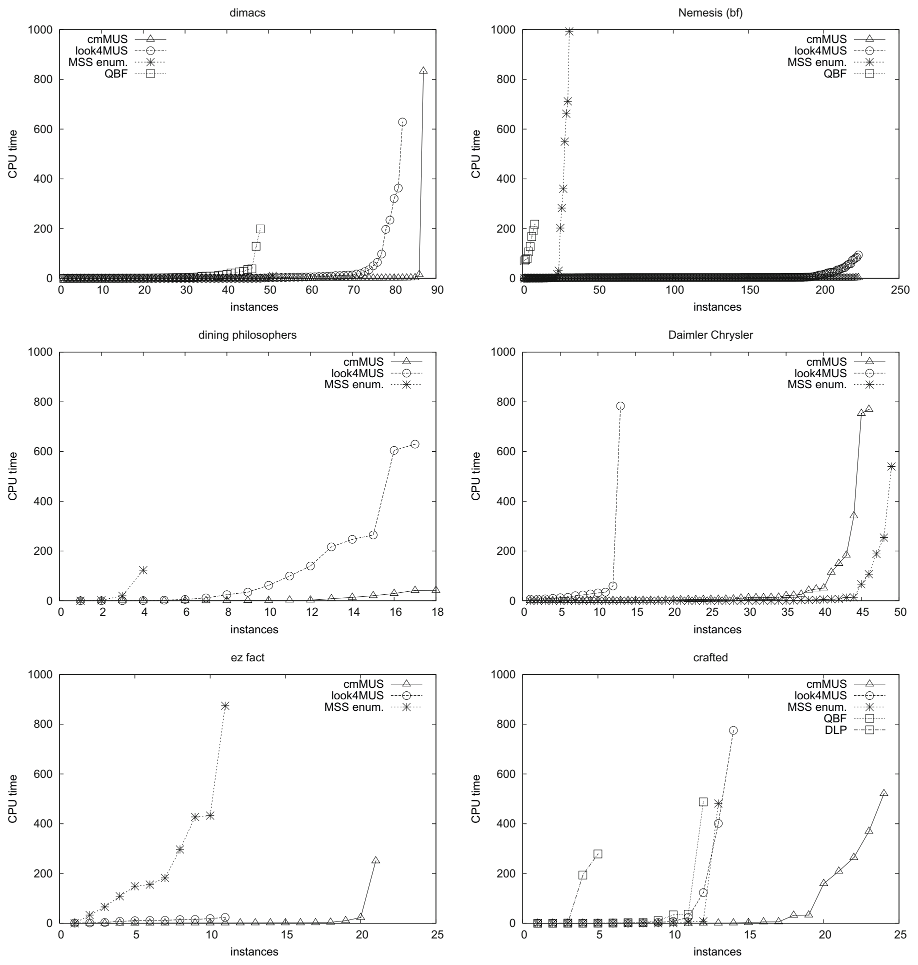


Fig. 1. Cactus plots for the measurements (number of instances x solved in less than y seconds)

5 Summary

This article presents cmMUS—a tool for deciding the MUS-MEMBERSHIP problem, i.e. it decides whether a given clause belongs to some minimally unsatisfiable set. The tool translates the problem into entailment in propositional circumscription, on which it invokes a dedicated algorithm based on abstraction counterexample refinement [9]. A variety of benchmarks shows that the tool outperforms existing approaches to the problem.

Acknowledgement. This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618), by FCT through grant ATTEST (CMU-PT/ELE/0009/2009), and by INESC-ID multiannual funding from the PIDDAC program funds.

References

1. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-driven disjunctive answer set solving. In: Brewka, G., Lang, J. (eds.) KR, pp. 422–432. AAAI Press, Menlo Park (2008)
2. Eiter, T., Gottlob, G.: Propositional circumscription and extended closed-world reasoning are π_2^{P}-complete. Theor. Comput. Sci. 114(2), 231–245 (1993)
3. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Annals of Mathematics and Artificial Intelligence 15, 289–323 (1995)
4. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3), 365–385 (1991)
5. Gelfond, M.: Answer Sets. In: Handbook of Knowledge Representation. Elsevier, Amsterdam (2008)
6. Giunchiglia, E., Marin, P., Narizzano, M.: An effective preprocessor for QBF pre-reasoning. In: 2nd International Workshop on Quantification in Constraint Programming, QiCP (2008)
7. Grégoire, É., Mazure, B., Piette, C.: Does this set of clauses overlap with at least one MUS? In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 100–115. Springer, Heidelberg (2009)
8. Janhunen, T., Oikarinen, E.: Capturing parallel circumscription with disjunctive logic programs. In: European Conf. on Logics in Artif. Intell., pp. 134–146 (2004)
9. Janota, M., Grigore, R., Marques-Silva, J.: Counterexample guided abstraction refinement algorithm for propositional circumscription. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 195–207. Springer, Heidelberg (2010)
10. Janota, M., Marques-Silva, J.: Models and algorithms for MUS membership testing. Tech. Rep. TR-07/2011, INESC-ID (January 2011)
11. Kullmann, O.: An application of matroid theory to the SAT problem. In: IEEE Conference on Computational Complexity, pp. 116–124 (2000)
12. Kullmann, O.: Constraint satisfaction problems in clausal form: Autarkies and minimal unsatisfiability. In: Electronic Colloquium on Computational Complexity (ECCC), vol. 14(055) (2007)
13. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning 40(1), 1–33 (2008)
14. McCarthy, J.: Circumscription - a form of non-monotonic reasoning. Artif. Intell. 13(1-2), 27–39 (1980)
15. Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential space. In: IEEE Conference Record of 13th Annual Symposium on Switching and Automata Theory (October 1972)
16. O'Callaghan, B., O'Sullivan, B., Freuder, E.C.: Generating corrective explanations for interactive constraint satisfaction. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 445–459. Springer, Heidelberg (2005)
17. Papadopoulos, A., O'Sullivan, B.: Relaxations for compiled over-constrained problems. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 433–447. Springer, Heidelberg (2008)

Transaction Logic with External Actions

Ana Sofia Gomes and José Júlio Alferes

Departamento de Informática
Faculdade Ciências e Tecnologias
Universidade Nova de Lisboa
2829-516 Caparica, Portugal

Abstract. We propose External Transaction Logic (or $\mathcal{ETR}$), an extension of Transaction Logic able to represent updates in internal and external domains whilst ensuring a relaxed transaction model. With this aim, $\mathcal{ETR}$ deals with two main components: an internal knowledge base where updates follow the strict ACID model, given by the semantics of Transaction Logic; and an external knowledge base of which one has limited or no control and can only execute *external* actions. When executing actions in the external domain, if a failure occurs, it is no longer possible to simply rollback to the initial state before executing the transaction. For dealing with this, similarly to what is done in databases, we define compensating operations for each external action to be performed to ensure a relaxed model of atomicity and consistency. By executing these compensations in backward order, we obtain a state considered to be equivalent to the initial one.

1 Introduction

Transaction Logic ($\mathcal{TR}$) is an extension of predicate logic proposed in [1] which exhibits a clean and declarative semantics, along with a sound and complete proof theory, to reason about state changes in arbitrary logical theories (as databases, logic programs or other knowledge bases). Unlike many other logic systems, $\mathcal{TR}$ imposes that the knowledge base evolves only into consistent states respecting ACID properties[1]. $\mathcal{TR}$ is parameterized by a pair of oracles that encapsulate elementary knowledge base operations of querying and updating, respectively, allowing $\mathcal{TR}$ to reason about elementary updates but without committing to a particular theory. Thus $\mathcal{TR}$ provides a logical foundation for knowledge base transactions accommodating a wide variety of semantics.

Example 1 (Financial Transactions). As illustration of $\mathcal{TR}$, consider a knowledge base of a bank [1] where the balance of an account is given by relation $balance(Acnt, Amt)$. To modify it we have a pair of elementary update operations: $balance(Acnt, Amt).ins$ and $balance(Acnt, Amt).del$ (denoting the insertion, resp. deletion, of a tuple of the relation). With these elementary updates, one may define several transactions, e.g. for making deposits in an account, make transfers from one account to another, etc. In $\mathcal{TR}$ one may define such transactions by the rules below where, e.g the first one means that one possible way to succeed the transfer of Amt from $Acnt$ to $Acnt'$ is by first withdrawing Amt from $Acnt$, followed by (denoted by $\otimes$) depositing Amt in $Acnt'$.

[1] As usual in databases, ACID stands for Atomicity, Consistency, Isolation and Durability.

J. Delgrande and W. Faber (Eds.): LPNMR 2011, LNAI 6645, pp. 272–277, 2011.

$$
\begin{array}{l}
transfer(Amt, Acnt, Acnt') \leftarrow withdraw(Amt, Acnt) \otimes deposit(Amt, Acnt') \\
withdraw(Amt, Acnt) \leftarrow balance(Acnt, B) \otimes changeBalance(Acnt, B, B - Amt) \\
deposit(Amt, Acnt) \leftarrow balance(Acnt, B) \otimes changeBalance(Acnt, B, B + Amt) \\
changeBalance(Acnt, B, B') \leftarrow balance(Acnt, B).del \otimes balance(Acnt, B').ins
\end{array}
$$

State change and evolution in $\mathcal{TR}$ is caused by executing ACID transactions, i.e. logical formulas posed into the system in a Prolog-like style as e.g. $? - transfer(10, a_1, a_2)$. Since every formula is assumed as a transaction, by posing $transfer(10, a_1, a_2)$ we know that either $transfer(10, a_1, a_2)$ can be executed respecting all ACID properties evolving the knowledge base from an initial state D_0 into a state D_n (passing through an arbitrary number of states n); or $transfer(10, a_1, a_2)$ cannot be executed under these conditions and so the knowledge base does not evolve and remains in the state D_0.

Unfortunately, $\mathcal{TR}$ is not suitable to model situations where besides executing ACID actions, some steps of the transaction require interaction with an external domain. That is, systems where the internal state of the knowledge base evolves ensuring the ACID properties, but this evolution depends on the execution of some actions in an external knowledge base, of which one has a limited control and interaction. This is, e.g., the case of web-based systems with an internal knowledge base that follows the strict ACID model, and that interact with other systems that they do not control, for instance, via web-services. As illustration, consider a system for a web shop that accepts orders from clients. Whenever a client submits an order, the system must take care of payments and updates of the inventory of the product to be sell. Obviously, it is crucial that each order is internally treated as a transaction. However, payments are validated and executed externally by the system of a bank, with which it communicates, but is external to the web shop, and over which it has a limited control.

Ensuring the standard ACID model in an external world is no longer possible. Particularly, the external actions executed in these domains cannot be rollbacked, as one has no control of the *external* system where they were performed. Moreover, since these actions require interaction with an external entity, this kind of transaction can last for relatively long periods of time, delaying the termination of shorter and more common transactions. To address this problem, [4] proposes the notion of long-lived transaction or sagas. The idea is to define compensating operations for each operation to be performed externally. If the transaction fails and these compensations are performed in backward order, then they lead the database into a state that is considered equivalent to the initial one, thus ensuring a weaker form of atomicity.

In this paper we propose External Transaction Logic ($\mathcal{ETR}$) that augments $\mathcal{TR}$ theory with the ability to reason about an external domain, and with a notion of compensations. The external reasoning is performed by an external oracle, parametric to the language, which describes the behavior of the external knowledge base. $\mathcal{ETR}$ allows for two different kinds of formulas: standard transaction formulas that follow the strict ACID model; and external action formulas, of the form $\mathbf{ext}(a, a^{-1})$, that follow a relaxed ACID model, and in case of failure of a executes the compensation a^{-1} leading the *external* knowledge base into an equivalent consistent state, but possibly different from the original one. As illustration, consider the following example:

Example 2. Consider the web shop mentioned above, where clients submits orders. In the end of each order, a final confirmation is asked to the client that may or not confirm the transaction. If the client accepts it, the order ends successfully. Otherwise, the transaction fails and the situation before its start must be reinstated. For this we need to rollback the update of the stock, and to compensate for the executed payment by asking the bank to refund the charged money. In $\mathcal{ETR}$ this can be modeled by:

$$\begin{aligned} buy(Prdt, Card, Amt) \leftarrow\ & \mathbf{ext}(chargeCard(Card, Amt), refundCard(Card, Amt)) \\ & \otimes updateStock(Prdt) \\ & \otimes \mathbf{ext}(confirmTransaction(Product, Card, Amt), ()) \end{aligned}$$

$$updateStock(Prdt) \quad \leftarrow N > 0 \otimes product(Prdt, N).del \otimes product(Prdt, N-1).ins$$

External actions can succeed or fail depending on the state of the external world. For instance, charging a given amount in a credit card depends on many things, e.g. if the credit limit is exceeded, or if the card has expired. To reason about the outcomes of these external actions $\mathcal{ETR}$ assumes the existence of an external oracle that comes as a parameter of the theory.

2 Syntax and Oracles

The theory of $\mathcal{TR}$ is parameterized by a pair of oracles $\mathcal{O}^d$ and $\mathcal{O}^t$, respectively denoted the data and the transition oracle. These oracles encapsulate the elementary knowledge base operations, allowing the separation of elementary operations from the logic of combining them. This separation allows $\mathcal{TR}$ to not commit to any particular theory of updates. Consequently, the language itself is not fixed and $\mathcal{TR}$ is able to accommodate a wide variety of knowledge base semantics, from classical to non-monotonic to various other non-standard logics [1]. $\mathcal{ETR}$ follows the same principles. In addition to the data oracle and the transition oracle, $\mathcal{ETR}$ requires an additional oracle to evaluate elementary external operations, the *external oracle*. Assuming this external oracle allows $\mathcal{ETR}$ to abstract the theory and semantics of the external domain, encapsulating the elementary operations that can be performed externally. These oracles are not fixed and almost any triple of oracles can be plugged into $\mathcal{ETR}$ theory.

To build complex logical formulas, $\mathcal{TR}$ uses the classical connectives $\wedge, \vee, \neg, \rightarrow$. In addition, $\mathcal{TR}$ also adds a new connective $\otimes$, denoted *serial conjunction* operator. Informally, the formula $\phi \otimes \psi$ represents an action composed of an execution of ϕ followed by an execution of ψ. Logical formulas in $\mathcal{ETR}$ are called *transaction formulas*, and a set of transaction formulas is called a *transaction base*. Furthermore, $\mathcal{ETR}$ extends $\mathcal{TR}$ with a special kind of formula $\mathbf{ext}(a, a^{-1})$ known as *external*. In this formula a and a^{-1} are atoms, where a denotes the action to be performed and a^{-1} its corresponding compensation. An $\mathcal{ETR}$ program is then defined as follows.

Definition 1 **($\mathcal{ETR}$ theories and programs).** *Given a language $\mathcal{L}$, a set of (internal) state identifiers $\mathcal{D}$ and a set of external state identifiers $\mathcal{E}$, an $\mathcal{ETR}$* theory *is a tuple $(T, \mathcal{O}^d, \mathcal{O}^t, \mathcal{O}^e)$ where T is a transaction base, and $\mathcal{O}^d$ (resp. $\mathcal{O}^t$; $\mathcal{O}^e$) is a mapping from elements in $\mathcal{D}$ (resp. pairs of elements in $\mathcal{D}$; elements in $\mathcal{E}$) into transaction formulas in the language of $\mathcal{L}$. An $\mathcal{ETR}$* program *consists of three parts: an $\mathcal{ETR}$ theory, an initial internal state identifier $\mathcal{D}_i$, and an initial external state identifier $\mathcal{E}_i$.*

3 Model Theory

The model theory for $\mathcal{ETR}$ is a generalization of $\mathcal{TR}$'s semantics, where the main novelty in $\mathcal{ETR}$ is the notion of *compensation*. A compensation occurs when the executed transaction ϕ contains external actions and fails. Since, in such a case, it is not possible to simply rollback to the initial state before executing ϕ, a series of compensating actions are executed to restore the consistency of the external knowledge base.

Central to $\mathcal{ETR}$'s model theory is the notion of paths, i.e. sequence of states. Logical formulas are evaluated on two sets of paths, of internal and of external states. Intuitively, external formulas of the form $\mathbf{ext}(a, a^{-1})$ are evaluated w.r.t. external paths by the external oracle, whereas the remaining logical formulas are evaluated in internal paths by the data and the transition oracle.

Definition 2 (Interpretations). *An interpretation is a mapping M that given a path of internal states, a path of external states and a sequence of actions, returns a set of transaction formulas (or $\top$)*[2]*. This mapping is subject to the following restriction:*

1. $\varphi \in M(\langle D\rangle, \langle E\rangle, \emptyset)$*, for every* φ *such that* $\mathcal{O}^d(D) \models \varphi$
2. $\varphi \in M(\langle D_1, D_2\rangle, \langle E\rangle, \emptyset)$ *if* $\mathcal{O}^t(D_1, D_2) \models \varphi$
3. $A \in M(\langle D\rangle, \langle E_1, \ldots, E_p\rangle, \langle A\rangle)$ *if* $\mathcal{O}^e(E_1, \ldots, E_p) \models A \quad p > 1$

The definition of satisfaction of more complex formulas, over general paths, requires the prior definition of operations on paths. These take into account how serial conjunction is satisfied, and how to construct the correct compensation.

Definition 3 (Paths and Splits). *A* path *of length k, or a* k-path*, is any finite sequence of states (where the Ss are all either internal or external states), $\pi = \langle S_1, \ldots, S_k\rangle$, where $k \geq 1$. A* split *of π is any pair of subpaths, π_1 and π_2, such that $\pi_1 = \langle S_1, \ldots, S_i\rangle$ and $\pi_2 = \langle S_i, \ldots, S_k\rangle$ for some i $(1 \leq i \leq k)$. In this case, we write $\pi = \pi_1 \circ \pi_2$.*

Definition 4 (External action split). *A sequence α of length j is any finite sequence of external actions (possibly empty), $\alpha = \langle A_1, \ldots, A_j\rangle$, where $j \geq 0$. A split of α is any pair of subsequences, α_1 and α_2, such that $\alpha_1 = \langle A_1, \ldots, A_i\rangle$ and $\alpha_2 = \langle A_{i+1}, \ldots, A_j\rangle$ for some i $(0 \leq i \leq k)$. In this case, we write $\alpha = \alpha_1 \circ \alpha_2$.*

Note that there is a significant difference between Definition 3 and Definition 4. In fact, splits for sequences of external actions can be empty, and particularly, it is also possible to define splits of empty sequences as $\emptyset = \emptyset \circ \emptyset$; whereas, a split of a path requires a sequence with at least length 1.

Definition 5 (Rollback split). *A rollback split of π is any pair of finite subpaths, π_1 and π_2, such that $\pi_1 = \langle D_1, \ldots, D_i, D_1\rangle$ and $\pi_2 = \langle D_1, D_{i+1}, \ldots, D_k\rangle$.*

Definition 6 (Inversion). *An external action inversion of a sequence α where $\alpha = (\mathbf{ext}(a_1, a_1^{-1}), \ldots, \mathbf{ext}(a_n, a_n^{-1}))$, denoted α^{-1}, is the corresponding sequence of compensating external actions performed in the inverse way as $(a_n^{-1}, \ldots, a_1^{-1})$.*

[2] Similar to $\mathcal{TR}$, for not having to consider partial mappings, besides formulas, interpretation can also return the special symbol $\top$.The interested reader is referred to [2] for details.

Definition 7 (Satisfaction). *Let M be an interpretation, π be an internal path, ϵ be an external path and α be a sequence of external actions. If $M(\pi, \epsilon, \alpha) = \top$ then $M, \pi, \epsilon, \alpha \models \phi$ for every transaction formula ϕ; otherwise:*

1. ***Base Case:*** *$M, \pi, \epsilon, \alpha \models p$ if $p \in M(\pi, \epsilon, \alpha)$ for any atomic formula p*
2. ***Negation:*** *$M, \pi, \epsilon, \alpha \models \neg\phi$ if it is not the case that $M, \phi, \epsilon, \alpha \models \phi$*
3. ***"Classical" Conjuction:*** *$M, \pi, \epsilon, \alpha \models \phi \wedge \psi$ if $M, \pi, \epsilon, \alpha \models \phi$ and $M, \pi, \epsilon, \alpha \models \psi$.*
4. ***Serial Conjuction:*** *$M, \pi, \epsilon, \alpha \models \phi \otimes \psi$ if $M, \pi_1, \epsilon_1, \alpha_1 \models \phi$ and $M, \pi_2, \epsilon_2, \alpha_2 \models \psi$ for some split $\pi_1 \circ \pi_2$ of path π, some split $\epsilon_1 \circ \epsilon_2$ of path ϵ, and some external action split $\alpha_1 \circ \alpha_2$ of external actions α.*
5. ***Compensating Case:*** *$M, \pi, \epsilon, \alpha \models \phi$ if $M, \pi_1, \epsilon_1, \alpha_1 \alpha_1^{-1} \rightsquigarrow \phi$ and $M, \pi_2, \epsilon_2, \alpha_2 \models \phi$ for some rollback split π_1, π_2 of π, some split $\epsilon_1 \circ \epsilon_2$ of path ϵ, and some external action split α_1, α_2 of α.*
6. *For no other $M, \pi, \epsilon, \alpha, \phi$, $M, \pi, \epsilon, \alpha \models \phi$.*

Satisfaction of disjunctions and implications are defined as usual, where $\phi \vee \psi$ means $\neg(\neg\phi \wedge \neg\psi)$, and $\phi \leftarrow \psi$ means $\phi \vee \neg\psi$.

The main novelty in this definition, when compared to $\mathcal{TR}$, is the inclusion of the compensation case. Intuitively, $M, \pi, \epsilon, \alpha, \alpha^{-1} \rightsquigarrow \phi$ means that, in the *failed* attempt to execute ϕ, a sequence α of external actions were performed. Since it is impossible to rollback to the point before the execution of these actions, consistency is ensured by performing a sequence of compensating external actions in backward order. Note that from the external oracle point of view there is no difference between a non-compensating external action and a compensating external action since both can fail or succeed, and in the latter case, evolving the external knowledge base. The path π represents the sequence of states consistent with the execution of α, but where $\pi = \langle D_1, \ldots, D_k, D_1 \rangle$, i.e. we explicitly rollback to the initial state, but keeping the trace of the failed evolution. We define a consistency preserving path $\pi, \epsilon, \alpha, \alpha^{-1}$ for a formula ϕ as follows.

Definition 8 (Consistency Preserving Path). *Let M be an interpretation, π be an internal path, ϵ an external path, and α be a* non-empty *sequence of external actions. The path π' is obtained from $\pi = \langle D_1, \ldots, D_n \rangle$ by removing the state D_n from the sequence; α^{-1} is a non-empty sequence of external actions obtained from α by inversion; ϵ_1 and ϵ_2 are some split of ϵ. We say that $M, \pi, \epsilon, \alpha, \alpha^{-1} \rightsquigarrow \phi$ iff $\exists b_1 \otimes \ldots \otimes b_i \otimes \ldots \otimes b_n$ such that:*
$M, \pi', \epsilon_1, \alpha \models \phi \leftarrow (b_1 \otimes \ldots \otimes b_i \otimes \ldots \otimes b_n)$
$M, \pi', \epsilon_1, \alpha \models b_1 \otimes \ldots \otimes b_i$
$M, \pi', \epsilon_1, \alpha \models \neg\, b_{i+1}$
$M, \pi', \epsilon_2, \alpha^{-1} \models \bigotimes \alpha^{-1}$
where $\bigotimes$ represents the operation of combining a sequence of actions using $\otimes$.

Note that there is no circularity in these definitions, as consistency preservation only appeals to satisfaction of formulae on strictly smaller internal paths. Also note that consistency preservation only applies to cases where α is not empty.

Definition 9 (Models). *An interpretation M is a model of a transaction formula ϕ if $M, \pi, \epsilon, \alpha \models \phi$ for every internal path π, every external path ϵ, and every action sequence α. In this case, we write $M \models \phi$. An interpretation is a model of a set of formulas if it is a model of every formula in the set.*

4 Conclusions

This work represents a first step towards a unifying logical framework able to combine a strict ACID transactions with long-running/relaxed model of transactions for hybrid evolving systems. That is, systems that have both an internal and an external updatable component and require properties on the outcomes of the updates. Examples of these systems range from an intelligent agent that has an internal knowledge base where he executes reasoning, but is integrated in an evolving external world where he can execute actions; to web-based systems with an internal knowledge base that follows the strict ACID model, but also need to interact with other systems, for instance to request a web-service. Closely related to this latter domain, is that of reactive (semantic) web languages. The Semantic Web initiative of W3C has recently proposed RIF-PRD [3] as a recommendation of a reactive (prodution-rule-like) rule language. This languages is intended to exchange rules that execute actions in hybrid web systems reactively, but still without concerns on guaranteeing transaction properties on the outcome of these actions. Given $\mathcal{ETR}$ declarative semantics and its natural model theory, we believe that it can be suitable to provide transaction properties for web-reactive systems. However, in order to make $\mathcal{ETR}$ suitable for these systems it is important to provide executional procedures to enable one to execute $\mathcal{ETR}$ transactions. This represents the next obvious step and is in line with what has been done in $\mathcal{TR}$.

Another important line of research is to further study the flexibility provided by having the oracles as a parameter of the theory. Particularly, how to take advantage of having as an external oracle logics that reason about state change or the related phenomena of time and action as as action languages [5], the situation calculus [8], event calculus [7], process logic [6] and many others. All these formalisms are orthogonal to $\mathcal{ETR}$, in the sense that they can just be "plugged" in the theory.

References

1. Bonner, A.J., Kifer, M.: Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, Computer Systems Research Institute, University of Toronto (1995)
2. Bonner, A.J., Kifer, M.: Results on reasoning about updates in transaction logic. In: Kifer, M., Voronkov, A., Freitag, B., Decker, H. (eds.) Dagstuhl Seminar 1997, DYNAMICS 1997, and ILPS-WS 1997. LNCS, vol. 1472, pp. 166–196. Springer, Heidelberg (1998)
3. de Sainte Marie, C., Hallmark, G., Paschke, A.: RIF Production Rule Dialect (June 2010), W3C Recommendation, `http://www.w3.org/TR/rif-prd/`
4. Garcia-Molina, H., Salem, K.: Sagas. SIGMOD Rec. 16, 249–259 (1987)
5. Gelfond, M., Lifschitz, V.: Action languages. Electr. Trans. Artif. Intell. 2, 193–210 (1998)
6. Harel, D., Kozen, D., Parikh, R.: Process logic: Expressiveness, decidability, completeness. In: FOCS, pp. 129–142 (1980)
7. Kowalski, R.A., Sergot, M.J.: A logic-based calculus of events. New Generation Comp. 4(1), 67–95 (1986)
8. McCarthy, J.: Situations, actions, and causal laws. Technical report, Stanford University (1963); Reprinted in MIT Press, Cambridge, pp. 410-417 (1968)

An Application of Clasp in the Study of Logics

Mauricio Osorio[2], José Luis Carballido[1], and Claudia Zepeda[1]

[1] Benemérita Universidad Atónoma de Puebla
[2] Universidad de las Américas - Puebla
{osoriomauri,jlcarballido7,czepedac}@gmail.com

Abstract. We show how to use the Answer Set Programming (ASP) tool called `clasp` to prove that there exists a unique three-valued paraconsistent logic that satisfies the substitution property and is sound with respect to da Costa C_ω logic.

Keywords: `clasp`, multi-valued logics, substitution property.

1 Introduction

In this paper, we describe and illustrate how to use answer set programming (ASP) to prove that there exists a unique three-valued paraconsistent logic, up to isomorphism, which extends the well known logic C_w and in which the substitution property is valid. This logic corresponds to G'_3, which has been studied in [9,5,10].

It is very useful to have software systems that help us to analyze logics. One of these systems is the ASP tool called `clasp` [6], which computes the answer sets of logic programs. ASP is a declarative knowledge representation and logic programming language based on the stable model semantics introduced in [7].

We take advantage of `clasp` since it allows us to define *redundant constraints* to define easily the primitive connectives as mathematical functions such as the $\vee$, $\neg$, $\wedge$, and $\rightarrow$. For instance, in `clasp` we write $1\{and(X,Y,Z) : v(Z)\}1 \leftarrow v(X), v(Y)$ instead of writing
$and(X,Y,Z) \leftarrow v(X), v(Z), v(Y), not\ nothera(X,Y,Z).$
$nothera(X,Y,Z) \leftarrow and(X,Y,Z1), Z! = Z1, v(X), v(Y), v(Z), v(Z1).$
as we wrote in a preliminary work in [12]. We also use the called *conditions* in `clasp` to define easily and briefly some constraints in our encoding. For instance, in `clasp` we write $\leftarrow not\ f(X) : not\ sel(X) : v(X)$ that helped us to express the property of paraconsistency.

In the past we already used ASP as a support system to find out properties of formal logics such as proving the independence of a set of axioms, proving that two logics are not equivalent, and to propose multivalued logics sound with respect to an axiomatic logic, see for instance [12].

It is worth to mention that the research in Paraconsistent logics is useful. Following Béziau [1], a logic is paraconsistent if it has a negation $\neg$, which is paraconsistent in the sense that $a, \neg a \nvdash b$, and at the same time has enough strong properties to be called a negation. da Costa et al. [4] mention applications of these logics in three different fields: Mathematics, Artificial Intelligence and Philosophy . An application that has not been fully recognized is the use of paraconsistent logics in non-monotonic reasoning. In this sense [5,10] illustrate such novel applications. Thus, the research on paraconsistent logics is far from being over.

Our paper is structured as follows. In section 2, we summarize some basic concepts and definitions necessary to understand this paper. In section 3, we show the `clasp`-encodings. Finally, in section 4, we present some conclusions.

2 Background

There are two ways to define a logic: by giving a set of axioms and specifying a set of inference rules; and by the use of truth values and interpretations. In this section we summarize each of them and we present some basic concepts and definitions useful to understand this paper.

2.1 Hilbert Style

In Hilbert style proof systems, also known as axiomatic systems, a logic is specified by giving a set of axioms and a set of inference rules, see [8]. In these systems, it is common to use the notation $\vdash_X F$ for provability of a logic formula F in the logic X. In that case we say that F is a theorem of X.

We say that a logic X is paraconsistent if the formula $(A \wedge \neg A) \rightarrow B$ is not a theorem[1].

A very important property satisfied by many logics is the substitution property which we present now.

Definition 1. *A logic X satisfies the substitution property if: $\vdash_X \alpha \leftrightarrow \beta$[2] then $\vdash_X \Psi[\alpha/p] \leftrightarrow \Psi[\beta/p]$ for any formulas α, β, and Ψ and any atom p that appear in Ψ where $\Psi[\alpha/p]$ denotes the resulting formula that is left after every occurrence of p is substituted by the formula α.*

As examples of axiomatic systems, we present two logics: the positive logic [9], and the C_ω logic which is a paraconsistent logic defined by da Costa [3]. In Table 1 we present a list of axioms, the first eight of them define positive logic. C_ω logic is defined by the axioms of positive logic plus axioms $\mathrm{C}_\omega1$ and $\mathrm{C}_\omega2$.

2.2 Multi-valued Logics

An alternative way to define a logic is by the use of truth values and interpretations. Multi-valued logics generalize the idea of using truth tables to determine

[1] For any logic X that contains Pos1 and Pos2 among its axioms and Modus Ponens as its unique inference rule, the formula $(A \wedge \neg A) \rightarrow B$ is a theorem if and only if $A, \neg A \vdash_X B$.

[2] Here we use the notation $\vdash_X$ to indicate that the formula that follows it is a theorem or a tautology depending on how the logic is defined.

Table 1. Axiomatization of C_ω

Pos1: $A \to (B \to A)$	C_ω**1:** $A \vee \neg A$
Pos2: $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$	C_ω**2:** $\neg\neg A \to A$
Pos3: $A \wedge B \to A$	
Pos4: $A \wedge B \to B$	
Pos5: $A \to (B \to (A \wedge B))$	
Pos6: $A \to (A \vee B)$	
Pos7: $B \to (A \vee B)$	
Pos8: $(A \to C) \to ((B \to C) \to (A \vee B \to C))$	

the validity of formulas in classical logic. The core of a multi-valued logic is its *domain* of values $\mathcal{D}$, where some of such values are special and identified as *designated* or `select` values. Logic connectives (e.g. $\wedge$, $\vee$, $\to$, $\neg$) are then introduced as operators over $\mathcal{D}$ according to the particular definition of the logic, see [8].

An *interpretation* is a function $I : \mathcal{L} \to \mathcal{D}$ that maps atoms to elements in the domain. The application of I is then extended to arbitrary formulas by mapping first the atoms to values in $\mathcal{D}$, and then evaluating the resulting expression in terms of the connectives of the logic (which are defined over $\mathcal{D}$). It is understood in general that, if I is an interpretation defined on the arbitrary formulas of a given program P, then $I(P)$ is defined as the function I applied to the conjunction of all the formulas in P. A formula F is said to be a *tautology*, denoted by $\models F$ if, for every possible interpretation, the formula F evaluates to a designated value. The simplest example of a multi-valued logic is classical logic where: $\mathcal{D} = \{0, 1\}$, 1 is the unique designated value, and the connectives are defined through the usual basic truth tables.

Note that in a multi-valued logic, so that it can truly be a *logic*, the implication connective has to satisfy the following property: for every value $x \in \mathcal{D}$, if there is a designated value $y \in \mathcal{D}$ such that $y \to x$ is designated, then x must also be a designated value. This restriction enforces the validity of Modus Ponens in the logic.

As an example of a multi-valued logic, we define G'_3, a logic that is relevant in this work.

The G'_3 logic is a 3-valued logic with truth values in the domain $D = \{0, 1, 2\}$ where 2 is the designated value. The evaluation functions of the logic connectives is then defined as follows: $x \wedge y = \min(x, y)$; $x \vee y = \max(x, y)$; $\neg\, x = 2$ if $x \leq 1$; and $\neg\, x = 0$ if $x = 2$. And $x \to y = 2$ if $x \leq y$, $x \to y = y$ if $x > y$.

3 Main Contribution

An interesting theoretical question that arises in the study of logics is whether a given logic satisfies the substitution property [11]. It is well known that there are several paraconsistent logics for which that theorem is not valid [2]. The encoding presented here helps us to prove that the only three-valued paraconsistent logic that extends C_w and satisfies the substitution property is G'_3.

Next we prove that any extension of C_ω that satisfies the weak substitution property satisfies also the substitution property.

Definition 2. *A logic X satisfies the weak substitution property if: $\vdash_X \alpha \leftrightarrow \beta$ then $\vdash_X \neg\alpha \leftrightarrow \neg\beta$.*

Theorem 1. *Any logic stronger than C_ω satisfies the weak substitution property iff satisfies the substitution property.*

Proof. One of the implications is immediate, the other one is done by induction on the size of formula Ψ. It only requires basic arguments valid in positive logic, except when the formula Ψ starts with a negation, in which case the weak substitution hypothesis is used.

The next theorem is the main part of the proof of the result stated at the beginning of this section, its proof is based on a `clasp`-encoding which includes the formula that defines the weak substitution property.

ASP has been used to develop different approaches in the areas of planning, logical agents and artificial intelligence. However, as far as the authors know, it has not been used as a tool to study logics. Here we use ASP to represent axioms and the inference rule Moduls Ponens in clasp in order to find three-valued logics that are paraconsistent and for which the axioms of C_ω are tautologies and the weak substitution property is valid.

Theorem 2. *G'_3 is the only three-valued paraconsistent logic, up to isomorphism, which extends C_ω and in which the weak substitution property is valid.*

Proof. Based on a set of values, and a subset of values corresponding to the property of being select, the `clasp`-encoding constructs the adequate truth tables for the connectives of a multi-valued logic that make all instances of the axioms of the given logic select. These truth tables are built in such a way that Modus Ponens preserves the property of being select. The encoding also includes the adequate conditions that each connective of the logic should satisfy, such as the arity, and the uniqueness; the definition of Modus Ponens; and all of the axioms of the logic. It is worth mentioning that all of the axioms are encoded as constraints. When each axiom is encoded as a constraint, the elimination of those assignment values of the logic connectives for which the axioms are not select, is guaranteed.

Now, we present the `clasp`-encoding, Π, that verifies our theorem. Due to lack of space, we only present the encoding of one of the axioms of C_ω, the encoding of the other axioms is similar. The encoding uses the values 0, 1, and 2 to create the truth tables for the connectives of the logic. The select value is 2. Thus, for any assignment of the values 0, 1 and 2 to the statements letters of a formula F, the tables determine a corresponding value for F. If F always takes the value 2, F will be called select. Furthermore, Π is a propositional program. As usual in ASP, we take for granted that programs with predicate symbols are only an abbreviation of the ground program. The encoding Π corresponds to the program $P_{val} \cup P_{prim} \cup P_{def} \cup P_{Ax} \cup P_{par} \cup P_{ws}$ that we present below. We want to remark that P_{def} includes all of the defined connectives of the C_ω logic, P_{Ax} includes the axioms of C_ω logic.

$$P_{val}: \left\{\begin{array}{l} \%\textbf{Truth values:}0,1,2 \\ v(0;1;2). \\ \%\textbf{Select value:}0 \\ sel(2). \end{array}\right. \qquad P_{prim}: \left\{\begin{array}{l} \%\textbf{Primitive connectives} \\ \%\textbf{implies, not, or, and, ...} \\ 1\{impl(X,Y,Z):v(Z)\}1 \leftarrow v(X), v(Y). \\ 1\{neg(X,Z):v(Z)\}1 \leftarrow v(X). \\ \ldots \end{array}\right.$$

$$P_{def}: \left\{\begin{array}{l} \%\textbf{Defined connectives} \\ \%\textbf{ if and only if, } \neg_{G_3}\textbf{, ...} \\ equ(X,Y,Z) \leftarrow impl(X,Y,L), impl(Y,X,R), and(L,R,Z). \\ \ldots \\ \%\textbf{Modus Ponens} \\ \leftarrow impl(X,Y,Z), sel(X), sel(Z), not\ sel(Y), v(X), v(Y), v(Z). \end{array}\right.$$

$$P_{Ax}: \left\{\begin{array}{l} \%\textbf{Axioms} \\ \%\textbf{A1}: A \to (B \to A) \\ \leftarrow impl(B,A,Z), impl(A,Z,R), v(R), not\ sel(R). \\ \ldots \end{array}\right.$$

$$P_{par}: \left\{\begin{array}{l} \%\textbf{Paraconsitency: } (A \wedge \neg A) \to B \text{ is not a theorem.} \\ eval(R) \leftarrow neg(A,A1), and(A,A1,L), impl(L,B,R), v(R). \\ \\ \leftarrow not\ eval(X) : not\ sel(X) : v(X). \end{array}\right.$$

$$P_{ws}: \left\{\begin{array}{l} \%\textbf{Weak substitution } if\ \vdash_X \alpha \leftrightarrow \beta,\ then\ \vdash_X \neg\alpha \leftrightarrow \neg\beta \\ \leftarrow equ(X,Y,Z), neg(X,X1), neg(Y,Y1), equ(X1,Y1,Z1), \\ \qquad\qquad v(Z), (Z1), sel(Z), not\ sel(Z1). \end{array}\right.$$

We can see that the weak substitution property is encoded as a constraint in P_{ws}. This means that in case of obtaining answer sets they must satisfy the weak substitution property. When we execute the `clasp`-encoding Π, we obtain two answer sets. Each of them corresponds to an adequate set of truth tables for the connectives of a paraconsistent three-valued logic that is sound with respect to the C_w logic and satisfies the weak substitution property. Moreover, these two sets of truth tables define isomorphic three-valued logics. Hence without loss of generality we obtain only one paraconsistent three-valued logic. In fact, it turns out that this three-valued logic is the G'_3 logic already introduced.

The proof shows that there is only one logic in the case we choose only one `select` value. We still need to look at the case where we consider two `select` values, but in this case there are no answer sets as the encoding shows. Hence, our proof is finished.

The following theorem is an immediate consequence of the last two theorems.

Theorem 3. *G'_3 is the only three-valued paraconsistent logic, up to isomorphism, which extends C_w and in which the substitution property is valid.*

4 Conclusions

ASP have been used to develop different approaches in the areas of planning, logical agents and artificial intelligence. However, as far as the authors know, it has not been used as a tool to study logics. We provide a `clasp`-encoding that can be used to obtain paraconsistent multi-valued logics and to verify the weak substitution property in a given logic.

References

1. Béziau, J.Y.: The paraconsistent logic Z. A possible solution to jaskowski's problem. Logic and Logical Philosophy 15, 99–111 (2006)
2. Carnielli, W.A., Marcos, J.: Limits for paraconsistent calculi. Notre Dame Journal of Formal Logic 40(3), 375–390 (1999)
3. da Costa, N.C.A.: On the theory of inconsistent formal systems. PhD thesis, Curitiva: Editora UFPR, Brazil (1963) (in Portuguese)
4. da Costa, N.C.A., Béziau, J.-Y., Bueno, O.A.S.: Aspects of paraconsistent logic. Logic Journal of the IGPL 3(4), 597–614 (1995)
5. Galindo, M.J.O., Ramírez, J.R.A., Carballido, J.L.: Logical weak completions of paraconsistent logics. J. Log. Comput. 18(6), 913–940 (2008)
6. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`, `http://potassco.sourceforge.net`
7. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Kowalski, R., Bowen, K. (eds.) 5th Conference on Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)
8. Mendelson, E.: Introduction to Mathematical Logic, 3rd edn. Wadsworth, Belmont (1987)
9. Osorio, M., Carballido, J.L.: Brief study of G'_3 logic. Journal of Applied Non-Classical Logic 18(4), 475–499 (2008)
10. Osorio, M., Navarro, J.A., Arrazola, J., Borja, V.: Logics with common weak completions. Journal of Logic and Computation 16(6), 867–890 (2006)
11. van Dalen, D.: Logic and Structure, 2nd edn. Springer, Berlin (1980)
12. Zepeda, C., Carballido, J.L., Marín, A., Osorio, M.: Answer set programming for studying logics. In: Special Session MICAI 2009, Puebla, México, pp. 153–158. IEEE Computer Society, Los Alamitos (2009), ISBN: 13 978-0-7695-3933-1

Industrial-Size Scheduling with ASP+CP

Marcello Balduccini

Kodak Research Laboratories
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

Abstract. Answer Set Programming (ASP) combines a powerful, theoretically principled knowledge representation formalism and powerful solvers. To improve efficiency of computation on certain classes of problems, researchers have recently developed hybrid languages and solvers, combining ASP with language constructs and solving techniques from Constraint Programming (CP). The resulting ASP+CP solvers exhibit remarkable performance on "toy" problems. To the best of our knowledge, however, no hybrid ASP+CP language and solver have been used in practical, industrial-size applications. In this paper, we report on the first such successful application, consisting of the use of the hybrid ASP+CP system EZCSP to solve sophisticated industrial-size scheduling problems.

1 Introduction

Answer Set Programming (ASP) [9,11] combines a powerful, theoretically principled knowledge representation formalism and efficient computational tools called solvers. In the ASP programming paradigm, a problem is solved by writing an ASP program that defines the problem and its solutions so that the program's models (or, more precisely, answer sets) encode the desired solutions. State-of-the-art ASP solvers usually allow to compute the program's models quickly. The paradigm makes it thus possible not only to use the language to study sophisticated problems in knowledge representation and reasoning, but also to quickly write prototypes, and even evolve them into full-fledged applications.

The growth in the number of practical applications of ASP in recent years has highlighted, and allowed researchers to study and overcome, the limitations of the then-available solvers. Especially remarkable improvements have been brought about by the cross-fertilization with other model-based paradigms. This in fact resulted in the integration in ASP solvers of efficient computation techniques from those paradigms. In CLASP [7], for example, substantial performance improvements have been achieved by exploiting clause learning and backjumping. Some researchers have also advocated the use of specialized solving techniques for different parts of the program. In fact, most ASP solvers operate on *propositional* programs – and are sensitive to the size of such programs – which may make computations inefficient when the domains of some predicates' arguments are large. This is particularly the case of programs with predicates whose arguments range over subsets of $\mathcal{N}$, $\mathcal{Q}$, or $\mathcal{R}$. On the other hand, this is a common situation in Constraint Programming (CP), and efficient techniques are available. For this reason, [3] proposed an approach in which ASP is extended with the ability to encode CP-style constraints, and the corresponding solver uses specialized techniques borrowed from CP solvers in handling such rules.

The research on integrating CP techniques in ASP has resulted in the development of various approaches, differing in the way CP constraints are represented in the hybrid language and in the way computations are carried out.

In [12] and [8], the language has been extended along the lines of [3], and specific ASP and CP solvers have been modified and integrated. In the former, syntax and semantics of ASP have been extended in order to allow representing, and reasoning about, quantities from $\mathcal{N}$, $\mathcal{Q}$, and $\mathcal{R}$. Separate solvers have been implemented for the different domains. In the latter, the focus has at least up to now been restricted to $\mathcal{N}$. The corresponding solver involves a remarkable integration of CP-solving techniques with clause learning and backjumping.

In [1], on the other hand, CP-style constraints have been encoded directly in ASP, without the need for extensions to the language. The corresponding representation technique allows dealing with quantities from $\mathcal{N}$, $\mathcal{Q}$, and $\mathcal{R}$, while the EZCSP solver allows selecting the most suitable ASP and CP solvers without the need for modifications. The current implementation of the solver supports $\mathcal{N}$ (more precisely, finite domains), $\mathcal{Q}$, and $\mathcal{R}$.

Although the above hybrid ASP+CP systems have shown remarkable performance on "toy" problems, to the best of our knowledge none of them has yet been confronted with practical, industrial-size applications. In this paper, we report on one such successful application, consisting in the use of EZCSP to solve sophisticated industrial-size scheduling problems. We hope that, throughout the paper, the reader will be able to appreciate how elegant, powerful, and elaboration tolerant the EZCSP formalization is.

The application domain of interest in this paper is that of industrial printing. In a typical scenario for this domain, orders for the printing of books or magazines are more or less continuously received by the print shop. Each order involves the execution of multiple jobs. First, the pages are *printed* on (possibly different) press sheets. The press sheets are often large enough to accommodate several (10 to 100) pages, and thus a suitable layout of the pages on the sheets must be found. Next, the press sheets are *cut* in smaller parts called *signatures.* The signatures are then *folded* into booklets whose page size equals the intended page size of the order. Finally the booklets are *bound* together to form the book or magazine to be produced. The decision process is made more complex by the fact that multiple models of devices may be capable of performing a job. Furthermore, many decisions have ramifications and inter-dependencies. For example, selecting a large press sheet would prevent the use of a small press. The underlying decision-making process is often called *production planning* (the term "planning" here is only loosely related to the meaning of planning the execution of actions over time typical in the ASP community, but is retained because it is relatively well established in the field of the application). Another set of decisions deals with *scheduling.* Here one needs to determine *when* the various jobs will be executed using the devices available in the print shop. Multiple devices of the same model may be available, thus even competing jobs may be run in parallel. Conversely, some of the devices can be offline – or go suddenly offline while production is in progress – and the scheduler must work around that. Typically, one wants to find a schedule that minimizes the tardiness of the orders while giving priority to the more important orders. Since orders are received on a continuous basis, one needs to be able to update the schedule in an incremental fashion, in a way that causes minimal disruption to the production, and can satisfy *rush orders*, which need to be executed quickly and take precedence over the others. Similarly, the scheduler needs to react to sudden changes in the print shop, such as a device going offline during production.

In this paper we describe the use of ASP+CP for the scheduling component of the system. It should be noted that the "toy" problems on which ASP+CP hybrids have been tested so far also include scheduling domains (see e.g. the 2nd ASP Competition [6]). However, as we hope will become evident later on, the constraints imposed on our system by practical use make the problem and the solution substantially more sophisticated, and the encoding and development significantly more challenging.

We begin by providing background on EZCSP. Next, we provide a general mathematical definition of the problem for our application domain. Later, we encode a subclass of problems of interest in EZCSP and show how schedules can be found by computing the extended answer sets of the corresponding encodings.

2 Background

In this section we provide basic background on EZCSP. The interested reader can find more details in [1]. Atoms and literals are formed as usual in ASP. A *rule* is a statement of the form $h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$, where h and l_i's are literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, has to believe h. A *program* is a set of rules. A rule containing variables is viewed as short-hand for the set of rules, called *ground instances*, obtained by replacing the variables by all possible ground terms. The ground instance of a program is the collection of the ground instances of its rules. Because of space considerations, we simply define an answer set of a program Π as one of the sets of brave conclusions entailed by Π under the answer set semantics. The precise definition can be found in [9]. Throughout this paper, readers who are not familiar with the definition can rely on the intuitive reading of ASP rules given above.

The definition of constraint satisfaction problem that follows is adapted from [13]. A *Constraint Satisfaction Problem (CSP)* is a triple $\langle X, D, C \rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is a set of domains, such that D_i is the domain of variable x_i (i.e. the set of possible values that the variable can be assigned), and C is a set of constraints. Each constraint $c \in C$ is a pair $c = \langle \sigma, \rho \rangle$ where σ is a list of variables and ρ is a subset of the

Cartesian product of the domains of such variables. An *assignment* is a pair $\langle x_i, a\rangle$, where $a \in D_i$, whose intuitive meaning is that variable x_i is assigned value a. A *compound assignment* is a set of assignments to distinct variables from X. A *complete assignment* is a compound assignment to all of the variables in X. A constraint $\langle \sigma, \rho\rangle$ specifies the acceptable assignments for the variables from σ. We say that such assignments *satisfy* the constraint. A *solution* to a CSP $\langle X, D, C\rangle$ is a complete assignment that satisfies every constraint from C. Constraints can be represented either *extensionally*, by specifying the pair $\langle \sigma, \rho\rangle$, or *intensionally*, by specifying an expression involving variables, such as $x < y$. In this paper we focus on constraints represented intensionally. A *global constraint* is a constraint that captures a relation between a non-fixed number of variables [10], such as $sum(x, y, z) < w$ and $all\_different(x_1, \ldots, x_k)$. One should notice that the mapping of an intensional constraint specification into a pair $\langle \sigma, \rho\rangle$ depends on the *constraint domain.* For this reason, in this paper we assume that every CSP includes the specification of the intended constraint domain.

In EZCSP, programs are written in such a way that their answer sets encode the desired CSPs. The solutions to the CSPs are then computed using a CP solver. CSPs are encoded in EZCSP using the following three types of statements: (1) a constraint domain declaration, i.e. a statement of the form $cspdomain(\mathcal{D})$, where $\mathcal{D}$ is a constraint domain such as fd, q, or r; informally, the statement says that the CSP is over the specified constraint domain (finite domains, $\mathcal{Q}$, $\mathcal{R}$), thereby fixing an interpretation for the intensionally specified constraints; (2) a constraint variable declaration, i.e. a statement of the form $cspvar(x, l, u)$, where x is a ground term denoting a variable of the CSP (CSP variable or constraint variable for short), and l and u are numbers from the constraint domain; the statement says that the domain of x is $[l, u]$;(3) a constraint statement, i.e. a statement of the form $required(\gamma)$, where γ is an expression that intensionally represents a constraint on (some of) the variables specified by the *cspvar* statements; intuitively the statement says that the constraint intensionally represented by γ is required to be satisfied by any solution to the CSP. For the purpose of specifying global constraints, we allow γ to contain expressions of the form $[\delta/k]$. If δ is a function symbol, the expression intuitively denotes the sequence of all variables formed from function symbol δ and with arity k, ordered lexicographically. If δ is a relation symbol, the expression intuitively denotes the sequence $\langle e_1, e_2, \ldots, e_n\rangle$ where e_i is the last element of the i^{th} k-tuple satisfying relation δ, according to the lexicographic ordering of such tuples.

Example 1. We are given 3 variables, v_1, v_2, v_3, ranging over $[1, 5]$ and we need to find values for them so that $v_2 - v_3 > 1$ and their sum is greater than or equal to 4. A possible encoding of the problem is $A_1 = \{cspdomain(fd), cspvar(v(1), 1, 5), cspvar(v(2), 1, 5), cspvar(v(3), 1, 5),$ $required(v(2) - v(3) > 1), required(sum([v/1]) \geq 4)\}$ $\diamond$

Let A be a set of atoms, including atoms formed from relations *cspdomain*, *cspvar*, and *required*. We say that A is a *well-formed CSP definition* if: (1) A contains exactly one constraint domain declaration; (2) the same CSP variable does not occur in two or more constraint variable declarations of A; and (3) every CSP variable that occurs in a constraint statement from A also occurs in a constraint variable declaration from A.

Let A be a well-formed CSP definition. The CSP *defined* by A is the triple $\langle X, D, C\rangle$ such that: (1) $X = \{x_1, x_2, \ldots, x_k\}$ is the set of all CSP variables from the constraint variable declarations in A; (2) $D = \{D_1, D_2, \ldots, D_k\}$ is the set of domains of the variables from X, where the domain D_i of variable x_i is given by arguments l and u of the constraint variable declaration of x_i in A, and consists of the segment between l and u in the constraint domain specified by the constraint domain declaration from A; (3) C is a set containing a constraint γ' for each constraint statement $required(\gamma)$ of A, where γ' is obtained by: (a) replacing the expressions of the form $[f/k]$, where f is a function symbol, by the list of variables from X formed by f and of arity k, ordered lexicographically; (b) replacing the expressions of the form $[r/k]$, where r is a relation symbol, by the sequence $\langle e_1, \ldots, e_n\rangle$, where, for each i, $r(t_1, t_2, \ldots, t_{k-1}, e_i)$ is the i^{th} element of the sequence, ordered lexicographically, of atoms from A formed by relation r; (c) interpreting the resulting intensionally specified constraint with respect to the constraint domain specified by the constraint domain declaration from A. A pair $\langle A, \alpha\rangle$ is an *extended answer set* of program Π iff A is an answer set of Π and α is a solution to the CSP defined by A.

Example 2. Set A_1 from Example 1 defines the CSP:

$$\langle\{v_1, v_2, v_3\}, \left\{\begin{array}{c}\{1,2,3,4,5\},\{1,2,3,4,5\},\\ \{1,2,3,4,5\}\end{array}\right\}, \left\{\begin{array}{c}v_2 - v_3 > 1,\\ sum(v(1), v(2), v(3)) \geq 4\end{array}\right\}\rangle.$$

3 Problem Definition

One distinguishing feature of ASP and derived languages is that they allow one to encode a problem at a level of abstraction that is close to that of the problem's mathematical (or otherwise precisely formulated, see e.g. [2]) specification. Consequently, it is possible for the programmer (1) to inspect specification and encoding and convince himself of the correctness of the encoding, and (2) to accurately prove the correctness of the encoding with respect to the formalization with more ease than using other approaches. *The ability to do this is quite important in industrial applications, where one assumes responsibility towards the customers for the behavior of the application.* Therefore, in this paper we precede the description of the encoding with a precisely formulated problem definition. Let us begin by introducing some terminology.

By *device class* (or simply *device*) we mean a type or model of artifact capable of performing some phase of production, e.g. a press model XYZ or a folder model MNO. By *device-set* of a print shop we mean the set of devices available in the shop. By *device instance* (or simply *instance*) we mean a particular exemplar of a device class. For example, a shop may have multiple XYZ presses, each of which is a device instance. We denote the fact that instance i is an instance of device class d by the expression $i \in d$.

A *job* j is a pair $\langle len, devices\rangle$ where len is the length of the job, and $devices \subseteq$ *device-set* is a set of devices that are capable of performing the job. Given a job j, the two components are denoted by $len(j)$ and $devices(j)$. Intuitively, the execution of a job can be split among a subset of the instances of the elements of $devices(j)$. A *job-set* J is a pair $\langle \Gamma, PREC\rangle$, where Γ is a set of jobs, and $PREC$ is a directed acyclic graph over the elements of Γ, intuitively describing a collection of precedences between jobs. An arc $\langle j_1, j_2\rangle$ in $PREC$ means that the execution of job j_1 must precede that of job j_2 (that is, the execution of j_1 must be completed before j_2 can be started). The components of J are denoted, respectively, by Γ_J and $PREC_J$.

The *usage-span* of an instance i is a pair $\langle start\text{-}time_i, duration_i\rangle$, intuitively stating that instance i will be in use (for a given purpose) from $start\text{-}time_i$ and for the specified duration. By US we denote the set of all possible usage-spans. Given a usage-span u, $start(u)$ denotes its first component and $dur(u)$ denotes its second. In the remainder of the discussion, given a partial function f, we use the notation $f(\cdot) = \perp$ (resp., $f(\cdot) \neq \perp$) to indicate that f is undefined (resp., defined) for a given tuple.

Definition 1 (Work Assignment). *A* work assignment *for a job-set J is a pair $\langle I, U\rangle$, where: (1) I is a function that associates to every job $j \in \Gamma_J$ a set of device instances (we use I_j as an alternative notation for $I(j)$), (2) U is a collection of (partial) functions* $\mathrm{usage_j} : I_j \to US$ *for every $j \in \Gamma_J$, and the following requirements are satisfied:*

1. *(I_j is valid) For every $i \in I_j$, $\exists d \in devices(j)$ s.t. $i \in d$.*
2. *($usage_j$ is valid) $\forall usage_j \in U$, $\sum_{i \in I_j, usage_j(i) \neq \perp} dur(usage_j(i)) = len(j)$.*
3. *(overlap) For any two jobs $j \neq j'$ from Γ_J and every $i \in I_j \cap I_{j'}$ such that $usage_j(i) \neq \perp$ and $usage_{j'}(i) \neq \perp$:*

$$start(usage_j(i)) + dur(usage_j(i)) \leq start(usage_{j'}(i)), \quad or$$
$$start(usage_{j'}(i)) + dur(usage_{j'}(i)) \leq start(usage_j(i)).$$

4. *(order) For every $\langle j, j'\rangle \in PREC_J$, j' starts only after j has been completed.*

$\diamond$

A *single-instance work assignment* for job-set J is a work assignment $\langle I, U\rangle$ such that $\forall j \in \Gamma_J, |I_j| = 1$. A single-instance work assignment intuitively prescribes the use of exactly one device instance for each job. In the rest of the discussion, we will focus mainly on a special type of job-set: a *simplified job-set* is a job-set J such that, for every job j from Γ_J, $|devices(j)| = 1$. Notice that work assignments for a simplified job-set are not necessarily single-instance. If multiple instances of some device are available, it is still possible to split the work for a job between the instances, or assign it to a particular instance depending on the situation. Next, we define various types of scheduling problems of interest and their solutions.

A *deadline-based decision (scheduling) problem* is a pair $\langle J, d\rangle$ where J is a set of jobs and d is a (partial) *deadline function* $d : J \to \mathcal{N}$, intuitively specifying deadlines for the jobs, satisfying the condition:

$$\forall j, j' \in \Gamma_J \quad [\ \langle j, j'\rangle \in PREC_J \to d(j) = \perp\ ]. \tag{1}$$

A *solution to a deadline-based decision problem* $\langle J, d\rangle$ is a work assignment $\langle I, U\rangle$ such that, for every $j \in \Gamma_J$: $\forall i \in I_j$ [$d(j) \neq \perp \rightarrow start(usage_j(i)) + dur(usage_j(i)) \leq d(j)$].

A *cost-based decision problem* is a tuple $\langle J, c, k\rangle$ where J is a job-set, c is a *cost function* that associates a cost (represented by a natural number) to every possible work assignment of J, and k is a natural number, intuitively corresponding to the target cost. A *solution to a cost-based decision problem* $\langle J, c, k\rangle$ is a work assignment W such that

$$c(W) \leq k. \tag{2}$$

An *optimization problem* is a pair $\langle J, c\rangle$ where J is a job-set and c is a *cost function* that associates a cost to every possible work assignment of J. A *solution to an optimization problem* $P = \langle J, c\rangle$ is a work assignment W such that, for every other work assignment W', $c(W) \leq c(W')$.

Since a solution to an optimization problem $\langle J, c\rangle$ can be found by solving the sequence of cost-based decision problems $\langle J, c, 0\rangle, \langle J, c, 1\rangle, \ldots$, in the rest of this paper we focus on solving decision problems. Details on how the optimization problem is solved by our system will be discussed in a longer paper.

In our system, scheduling can be based on total tardiness, i.e. on the sum of the amount of time by which the jobs are past their deadline. Next, we show how this type of scheduling is an instance of a cost-based decision problem.

Example 3. Total tardiness
We are given a job-set J, the target total tardiness k, and a deadline function d (as defined earlier). We construct a cost function c so that the value of $c(W)$ is the total tardiness of work assignment W. The construction is as follows. First we define auxiliary function $c'(j, W)$ which computes the tardiness of job $j \in \Gamma_J$ based on W:

$$c'(j, W) = \begin{cases} 0 \quad \text{if } d(j) = \perp \text{ or } usage_j(i) = \perp \\ \max(0, start(usage_j(i)) + dur(usage_j(i)) - d(j)) \quad \text{otherwise.} \end{cases}$$

Now we construct the cost function as: $c(W) = \sum_{j \in \Gamma_J} c'(j, W)$. The work assignments with total tardiness less than or equal to some value k are thus the solutions of the cost-based decision problem $\langle J, c, k\rangle$. ◇

At this stage of the project,we focus on simplified job-sets and single-instance solutions. Furthermore, all instances of a given device are considered to be identical (one could deal with differences due e.g. to aging of some instances by defining different devices).

Under these conditions, a few simplifications can be made. Because we are focusing on single-instance solutions, given a work assignment $\langle I, U\rangle$, it is easy to see that I_j is a singleton for every j. Moreover, since we are also focusing on simplified job-sets, for every j, $usage_j$ is defined for a single device instance of a single device $d \in devices(j)$. It is not difficult to see that, for the usage-span $\langle start\text{-}time, duration\rangle$ of every job j, $duration = len(j)$. Thus, the only information that needs to be specified for work assignments is the start time of each job j and the device instance used for the job.

A solution to a scheduling problem can now be more compactly described as follows. A *device schedule* for d is a pair $\langle L, S \rangle$, where $L = \langle j_1, \ldots, j_k \rangle$ is a sequence of jobs, and $S = \langle s_1, \ldots, s_k \rangle$ is a sequence of integers. Intuitively, L is the list of jobs that are to be run on device d, and S is a list of start times such that each s_m is the start time of job j_m. A *global schedule* for a set of devices D is a function σ that associates each device from D with a device schedule.

4 Encoding and Solving Scheduling Problems

In this section, we describe how scheduling problems for our application domain are encoded and solved using EZCSP. Although our formalization is largely independent of the particular constraint domain chosen, for simplicity we fix the constraint domain to be that of finite domains, and assume that any set of rules considered also contains the corresponding specification $cspdomain(fd)$.

A device d with n instances is encoded by the set of rules $\varepsilon(d) = \{device(d). \quad instances(d,n).\}$. A job j is encoded by $\varepsilon(j) = \{job(j). \quad job\_len(j,l). \quad job\_device(j,d).\}$, where $l = len(j)$ and $d \in device(j)$ ($device(j)$ is a singleton under the conditions stated earlier).

The components of a job-set $J = \langle \Gamma, PREC \rangle$ are encoded as follows:

$$\varepsilon(\Gamma) = \bigcup_{j \in \Gamma_J} \varepsilon(j) \;\; ; \;\; \varepsilon(PREC_J) = \{precedes(j,j'). \mid \langle j,j' \rangle \in PREC_J\}.$$

The encoding of job-set J is $\varepsilon(J) = \varepsilon(\Gamma_J) \cup \varepsilon(PREC_J)$.

Given a scheduling problem P, the overall goal is to specify its encoding $\varepsilon(P)$. In this section we focus on solving cost-based decision problems $\langle J, c, k \rangle$ with c based on total tardiness, and defined as in Example 3. Our goal then is to provide the encoding $\varepsilon(\langle J, c, k \rangle)$.

We represent the start time of the execution of job j on some instance of device d by constraint variable $st(d,j)$. The definition of the set of such constraint variables is given by ε_{vars}, consisting of the rule:

$$cspvar(st(D,J),0,MT) \leftarrow job(J),\ job\_device(J,D),\ max\_time(MT).$$

together with the definition of relation max_time, which determines the upper bound of the range of the constraint variables. The next constraint ensures that the start times satisfy the precedences in $PREC_J$:

$$\varepsilon_{prec} = \left\{ \begin{array}{l} required(st(D2,J2) \geq st(D1,J1) + Len1) \leftarrow \\ \qquad job(J1),\ job(J2),\ job\_device(J1,D1),\ job\_device(J2,D2), \\ \qquad precedes(J1,J2),\ job\_len(J1,Len1). \end{array} \right.$$

The deadlines of all jobs $j \in \Gamma_J$ are encoded by a set ε_{dl} of facts of the form $deadline(j,n)$. One can ensure that requirement (1) is satisfied by specifying a constraint $\leftarrow precedes(J1,J2),\ deadline(J1,D)$. Function c' from Example 3 is encoded by introducing an auxiliary constraint variable $td(j)$ for every job j, where $td(j)$ represents the value of $c'(j, W)$ for the current work assignment. The encoding of c', $\varepsilon(c')$, consists of $\varepsilon_{var} \cup \varepsilon_{prec} \cup \varepsilon_{dl}$ together with:

$$\begin{array}{l} cspvar(td(J), 0, MT) \leftarrow job(J),\ max\_time(MT). \\ required(td(J) == max(0, st(D, J) + Len - Deadline)) \leftarrow \\ \qquad job(J),\ job\_device(J, D),\ deadline(J, Deadline),\ job\_len(J, Len). \end{array}$$

Notice that the constraint is only enforced if a deadline has been specified for job j. An interesting way to explicitly set the value of $td(j)$ to 0 for all other jobs consists in using:

$$\begin{array}{l} enforced(td(J)) \leftarrow job(J),\ required(td(J) == X),\ X \neq 0. \\ required(td(J) == 0) \leftarrow job(J),\ \text{not}\ enforced(td(J)). \end{array}$$

Function c from Example 3 is encoded by introducing an auxiliary constraint variable tot_tard. The encoding, $\varepsilon(c)$, consists of $\varepsilon(c')$ together with:

$$\begin{array}{l} cspvar(tot\_tard, 0, MT) \leftarrow max\_time(MT). \\ required(sum([td/1], ==, tot\_tard)). \end{array}$$

where the constraint intuitively computes the sum of all constraint variables $td(\cdot)$. The final step in encoding the decision problem $\langle J, c, k \rangle$ is to provide a representation, $\varepsilon(k)$, of constraint (2), which is accomplished by the rule:

$$required(tot\_tard \leq K) \leftarrow max\_total\_tardiness(K).$$

together with the definition of relation $max\_total\_tardiness$, specifying the maximum total tardiness allowed. It is interesting to note the flexibility of this representation: if relation $max\_total\_tardiness$ is not defined, then the above constraint is not enforced – in line with the informal reading of the rule – and thus one can use the encoding to find a schedule irregardless of its total tardiness.

The encoding of a cost-based decision problem $\langle J, c, k \rangle$, where c computes total tardiness, is then: $\varepsilon(\langle J, c, k \rangle) = \varepsilon(J) \cup \varepsilon(c) \cup \varepsilon(k)$.

Now that we have a complete encoding of the problem, we discuss how scheduling problems are solved. Given $\varepsilon(\langle J, c, k \rangle)$, to solve the scheduling problem we need to assign values to the constraint variables while enforcing the following requirements:

[Overlap] if two jobs j and j', being executed on the same device, overlap (that is, one starts before the other is completed), then they must be executed on two separate instances of the device;

[Resources] at any time, no more instances of a device can be used than are available.

This can be accomplished compactly and efficiently using global constraint *cumulative* [4]. The *cumulative* constraint takes as arguments: (1) a list of constraint variables encoding start times; (2) a list specifying the execution length of each job whose starts time is to be assigned; (3) a list specifying the amount of resources required for each job whose start time is to be assigned; (4) the maximum number of resources available at any time on the device. In order to use *cumulative*, we represent the number of available device instances as an amount of resources, and use a separate cumulative constraint for the schedule of each device. The list of start times for the jobs that are to be processed by device d can be specified, in EZCSP, by a term $[st(d)/2]$, intuitively denoting the list of terms (1) formed by function symbol st, (2) of arity 2, and (3) with d as first argument. We also introduce auxiliary relations *len_by_dev* and *res_by_dev*, which specify, respectively, the length and number of resources for the execution on device d of job j. The auxiliary relations are used to specify the remaining two lists for the *cumulative* constraint, using EZCSP terms $[len\_by\_dev(D)/3]$ and $[res\_by\_dev(D)/3]$. The complete scheduling module, Π_{solv}, is:

$$required(cumulative([st(D)/2], [len\_by\_dev(D)/3], [res\_by\_dev(D)/3], N)) \leftarrow \\ instances(D, N).$$

$$len\_by\_dev(D, J, N) \leftarrow job(J),\ job\_device(J, D),\ job\_len(J, N).$$

$$res\_by\_dev(D, J, 1) \leftarrow job(J),\ job\_device(J, D).$$

Notice that, since we identify the resources with the instances of a device, the amount of resources required by any job j at any time is 1.

Schedules for a cost-based decision problem $\langle J, c, k\rangle$, where c computes total tardiness, can be then found by computing the extended answer sets of the program $\varepsilon(\langle J, c, k\rangle) \cup \Pi_{solv}$. Using the definitions from Section 3, one can prove the following:

Proposition 1. *Let $\langle J, c, k\rangle$ be a cost-based decision problem, where c computes total tardiness. The global schedules of $\langle J, c, k\rangle$ are in one-to-one correspondence with the extended answer sets of the program $\varepsilon(\langle J, c, k\rangle) \cup \Pi_{solv}$.*

5 Incremental and Penalty-Based Scheduling

In this section, we describe the solution to more sophisticated scheduling problems. To give more space to the encoding, we omit the precise problem definitions, which can be obtained by extending the definitions from Section 3.

It is important to stress that *these extensions to the scheduler are entirely incremental, with no modifications needed to the encoding from Section 4*. This remarkable and useful property is the direct result of the elaboration tolerance typical of ASP encodings.

The first extension of the scheduler consists in considering a more sophisticated cost-based decision problem, $\langle J, c^*, k^*\rangle$. In everyday use, some orders take precedence over others, depending on the service level agreement the shop has with its customers. Each service level intuitively yields a different penalty if the corresponding jobs are delivered late. The total penalty of a schedule is thus obtained as the weighted sum of the tardiness of the jobs, where the weights are based on each job's service level. The encoding, $\varepsilon(c^*)$, of c^* consists of $\varepsilon(c)$ together with (relation *weight* defines the assignments of weights to jobs):

$$cspvar(penalty(J), 0, MP) \leftarrow job(J),\ max\_penalty(MP).$$
$$required(penalty(J) == td(J,O) * Weight) \leftarrow job(J),\ weight(J, Weight).$$
$$cspvar(tot\_penalty, 0, MP) \leftarrow max\_penalty(MP).$$
$$required(sum([penalty/2], ==, tot\_penalty)).$$

The encoding of $\varepsilon(k^*)$ extends $\varepsilon(k)$ by the rule:

$$required(tot\_penalty \leq P) \leftarrow max\_total\_penalty(P).$$

which encodes constraint (2) for penalties. Notice that, thanks to the elaboration tolerance of the encoding developed in Section 4, it is safe for $\varepsilon(k^*)$ to include $\varepsilon(k)$, since now relation *max_total_tardiness* is left undefined.

Another typical occurrence in everyday use is that the schedule must be updated incrementally, either because new orders were received, or because of equipment failures. During updates, intuitively one needs to avoid re-scheduling jobs that are already being executed. We introduce a new decision problem $\langle J^*, c^*, k^* \rangle$, where c^*, k^* are as above, and J^* extends J to include information about the current schedule; $\varepsilon(J^*)$ includes $\varepsilon(J)$ and is discussed next.

To start, we encode the current schedule by relations $curr\_start(j, t)$ and $curr\_device(j, d)$. The current (wall-clock) time t is encoded by relation $curr\_time(t)$. The following rules informally state that, if according to the current schedule production of a job has already started, then its start time and device must remain the same. Conversely, all other jobs must have a start time that is no less than the current time[1].

$$already\_started(J) \leftarrow curr\_start(J,T),\ curr\_time(CT),\ CT > T.$$
$$must\_not\_schedule(J) \leftarrow\ already\_started(J),\ \text{not}\ ab(must\_not\_schedule(J)).$$
$$required(st(D,J) \geq CT) \leftarrow\ job\_device(J,D),\ curr\_time(CT),\ \text{not}\ must\_not\_schedule(J).$$
$$required(st(D,J) == T) \leftarrow\ curr\_device(J,D),\ curr\_start(J,T),\ must\_not\_schedule(J).$$

It is worth noting that the use of a default to define *must_not_schedule* allows extending the scheduler in a simple and elegant way by defining exceptions.

Whereas the above rules allow scheduling new jobs without disrupting current production, the next set deals with equipment going offline, even during production. Notice that by "equipment" we mean a particular instance of a device. To properly react to this situation, the system first needs to have an explicit representation of which device instance is assigned to perform which job. This can be accomplished by introducing a new variable $on\_instance(j)$. The value of the variable is a number that represents the device instance assigned to the job (among the instances of the device prescribed by $job\_device(j, d)$). The corresponding variable declaration and constraints are encoded by:

[1] One may not want to schedule jobs to start exactly at the current time, as there would not be time to move the supplies and set up the job. Extending the rules to accomplish that is straightforward.

$$
\begin{aligned}
&cspvar(on\_instance(J), 1, N) \leftarrow job\_device(J, D),\ instances(D, N).\\
&required((on\_instance(J1) \neq on\_instance(J2))\ \lor\\
&\qquad (st(D, J2) \geq st(D, J1) + Len1)\ \lor\ (st(D, J1) \geq st(D, J2) + Len2)) \leftarrow\\
&\qquad job\_device(J1, D),\ job\_device(J2, D),\ J1 \neq J2,\\
&\qquad len(J1, Len1),\ len(J2, Len2),\ instances(D, N),\ N > 1.\\
&required(on\_instance(J) \neq I) \leftarrow\\
&\qquad job\_device(J, D),\ offline\_instance(D, I),\ \text{not}\ must\_not\_schedule(J).
\end{aligned}
$$

The second rule intuitively says that, if d has more than one instance and is scheduled to run two (or more) jobs, then either a job ends before the other starts, or the jobs must be assigned to two different instances. The rule is an example of use of *reified constraints*, as defined for example in [5]. The last rule says that no job can be assigned to an offline instance. For incremental scheduling, we also extend the encoding of the current schedule by introducing a relation $curr\_on\_instance(j, i)$, which records the instance-job previously determined.

The reader might wonder about the overlap between *cumulative*, used earlier, and the above constraints. In fact, *cumulative* already performs a limited form of reasoning about instance-job assignments when it determines if sufficient resources are available to perform the jobs. Although it is possible to replace *cumulative* by a set of specific constraints for the assignment of start times, we believe that using *cumulative* makes the encoding more compact and readable.

The encoding $\varepsilon(J^*)$ is completed by rules that detect situations in which a device went offline while executing a job. Relation $offline\_instance(d, i)$ states that instance i of device d is currently offline.

$$
\begin{aligned}
&already\_finished(J) \leftarrow\ curr\_start(J, T),\ len(J, Len),\ curr\_time(CT),\ CT \geq T + Len.\\
&ab(must\_not\_schedule(J)) \leftarrow\\
&\qquad already\_started(J),\ \text{not}\ already\_finished(J),\ curr\_device(J, D),\\
&\qquad curr\_on\_instance(J, I),\ offline\_instance(D, I).
\end{aligned}
$$

The last rule defines an exception to the default introduced earlier. Informally, the rule says that if instance i is offline, any job assigned to it that is currently in production constitutes an exception to the default. It is not difficult to see that such exceptional jobs are subjected to regular rescheduling. The presence of atoms formed by relation ab in the extended answer set is also used by the system to warn the user that a reschedule was forced by a device malfunction.

In conclusion, the solutions to problem $\langle J^*, c^*, k^* \rangle$ can be found by computing the extended answer sets of $\varepsilon(\langle J^*, c^*, k^* \rangle)$.

6 Conclusions

In this paper we have described what to the best of our knowledge is the first industrial-size application of an ASP+CP hybrid language. The application is currently being considered for use in commercial products. Performance evaluation of our system is under way. A simplified version of the domain has been accepted as a benchmark for the Third ASP Competition at LPNMR-11. Preliminary analysis on customer data showed that performance is comparable to that of similar implementations written using CP alone, *with schedules for customer-provided instances typically found in less than one minute, and often in a few seconds*. In comparison with direct CP encodings, we found that the compactness, elegance, and elaboration tolerance of the EZCSP encoding are superior. In a CLP implementation that we have developed for comparison, the number of rules in the encoding was close to one order of magnitude larger than the number of rules in the EZCSP implementation. Moreover, writing those rules often required one to consider issues with procedural flavor, such as how and where certain information should be collected.

References

1. Balduccini, M.: Representing Constraint Satisfaction Problems in Answer Set Programming. In: ICLP 2009 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2009) (July 2009)
2. Balduccini, M., Girotto, S.: Formalization of Psychological Knowledge in Answer Set Programming and its Application. Journal of Theory and Practice of Logic Programming (TPLP) 10(4-6), 725–740 (2010)
3. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an Integration of Answer Set and Constraint Solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
4. Beldiceanu, N., Contejean, E.: Introducing Global Constraints in CHIP. Mathl. Comput. Modelling 20(12), 97–123 (1994)
5. Carlson, B., Carlsson, M., Ottosson, G.: An Open-Ended Finite Domain Constraint Solver. In: Hartel, P.H., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292. Springer, Heidelberg (1997)
6. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Veloso, M.M. (ed.) Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 386–392. MIT Press, Cambridge (2007)
8. Gebser, M., Ostrowski, M., Schaub, T.: Constraint Answer Set Solving. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 235–249. Springer, Heidelberg (2009)
9. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
10. Katriel, I., van Hoeve, W.J.: Global Constraints. In: Handbook of Constraint Programming, ch. 6. Foundations of Artificial Intelligence, pp. 169–208. Elsevier, Amsterdam (2006)
11. Marek, V.W., Truszczynski, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer, Berlin (1999)
12. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating Answer Set Programming and Constraint Logic Programming. Annals of Mathematics and Artificial Intelligence (2008)
13. Smith, B.M.: Modelling. In: Handbook of Constraint Programming, ch. 11. Foundations of Artificial Intelligence, pp. 377–406. Elsevier, Amsterdam (2006)

Secommunity: A Framework for Distributed Access Control

Steve Barker[1] and Valerio Genovese[2,3]

[1] King's College London
[2] University of Luxembourg
[3] University of Torino

Abstract. We describe an approach for distributed access control policies that is based on a nonmonotonic semantics and the use of logic programming for policy specification and the evaluation of access requests. Our approach allows assertions of relevance to access control to be made by individual agents or on a community-based level and different strengths of testimonial warrant may be distinguished by using various logical operators. We describe a form of ASP that allows for remote access request evaluation and we discuss a DLV-based implementation of our approach.

1 Introduction

We address an aspect of a basic form of "the" access control problem, of specifying and deciding who (which principals) have what type of access privileges on what resources.

In this paper, our focus is on a nonmonotonic approach to access control in distributed computing contexts. In this case, the specification of policies and answers to requests, by principals to access a resource, often involve representing and computing with incomplete forms of information. For example, a principal identified by the key K_α may request read access to K_β's "special offers" file but without K_β necessarily knowing enough itself about K_α to determine, with confidence, what response to provide to K_α's request. Nonmonotonic reasoning is important in this context. In our example scenario, an atomic consequence like $authorized(K_\alpha, read, special\ offers)$ may need to be retracted if additional information becomes known about K_α. Similarly, a "standard" certificate, used in distributed access control, has the semantics [6]: "This certificate is good until the expiration date. Unless, of course, you hear that it has been revoked." The "unless" part highlights an inherent nonmonotonicity. The expiration of a certificate's validity period may also be viewed as being "temporally nonmonotonic" [4].

The novelty of our contribution is to be understood in terms of our description, in logic programming terms, of a community view of socially constructed testimonial warrant and the adoption of a coherence-based semantics, i.e., a community of asserters reach a coherent view on the assertions made by a community.

We adopt a conceptual model that includes has four main types of entities: *resources, acceptors, oracles and contributors*. Assertions are contributed by community members to oracles that are then used by acceptors, of an oracle's assertions, to evaluate requests to access resources that are controlled by the acceptors. Oracles are repositories that store community assertions; oracles are used by a community as long as they serve the community in terms of making assertions that are valuable in deciding access requests. The principals that request access to resources that are protected by acceptors may be viewed as a fifth key sort of agent in our our framework. The approach that we describe in this paper also extends our previous work on meta-models for access control [1], which is based on exploiting the notion of a category (see below) for developing a unified view of access control. The main focus in this work is to describe a novel ASP-based implementation of our approach that makes use of externally defined predicates.

The rest of this paper is organized thus. In Section 2, we describe the foundational details on which our approach is based. In Section 3, we consider implementation details. In Section 4, conclusions are drawn and further work is suggested.

2 Access Control Policies in ASP

In this section, we briefly introduce the key syntactic and semantic concepts that we use. The main sorts of constants in the (non-empty) universe of discourse that we assume are: A countable set $\mathcal{C}$ of categories, where c_0, c_1, $\ldots$ are (strings) used to denote arbitrary category identifiers; A countable set $\mathcal{P}$ of principals, where p_0, p_1, $\ldots$ are used to identify users, organizations, processes, ...; A countable set $\Sigma \subseteq \mathcal{P}$ of *sources* of testimonial warrant, where $s_0, s_1, \ldots$ are (strings) used to denote arbitrary sources; A countable set $\mathcal{A}$ of named atomic *actions*, where $a_0, a_1, \ldots$ are (strings) used to denote arbitrary action identifiers; A countable set $\mathcal{R}$ of *resource identifiers*, where $r_0, r_1, \ldots$ denote arbitrary resources; $r(t_1, \ldots, t_n)$ is an arbitrary n-place relation that represents an "information resource" where t_i $(1 \leq i \leq n)$ is a term (a term is a function, a constant or a variable).

Informally, a category is any of several fundamental and distinct classes or groups to which entities may be assigned (cf. [1]). The categories of interest, for what we propose, are application-specific and determined by usage (individual, community or universal) rather than by necessary and sufficient conditions. A community will define categories in terms of a chosen vocabulary. Named categories like $trusted$, $highly\ trusted$, $untrusted$, $\ldots$ may be community accepted terms or each community member can define its own categories, which may then be referred to within the community. In all cases, the community decides the language it uses (cf. "vocabulary agreements" [5]). Our adoption of a community-based vocabulary is consistent with our view of community-based assertion-making.

The language in terms of which we specify access control policies is centered on four theory-specific predicates, pca, $pcas$, $arca$ and par, which have following meanings: $pca(p, c)$ iff the principal $p \in \mathcal{P}$ is assigned to the category $c \in \mathcal{C}$; $pcas(p, c, s)$ iff the member $s \in \Sigma$ asserts that the principal $p \in \mathcal{P}$ is assigned to the category $c \in \mathcal{C}$; $arca(a, r, c)$ iff the privilege of performing action $a \in \mathcal{A}$ on resource $r \in \mathcal{R}$ is assigned to the category $c \in \mathcal{C}$; $par(p, a, r)$ iff the principal $p \in \mathcal{P}$ is authorized to perform the action $a \in \mathcal{A}$ on resource $r \in \mathcal{R}$. The extension of par is defined in terms of $arca$ and pca by the following rule: $\forall p \forall a \forall r \forall c((arca(a, r, c) \wedge pca(p, c)) \rightarrow par(p, a, r))$ which reads as: "If a principal p is assigned to a category c to which the privilege to perform action a on resource r has been assigned, then p is authorized to perform the a action on r".

In addition to pca, $pcas$, $arca$ and par, we add "counting operators" ranging over the pca predicate and with a semantics that may be informally understood as follows: $\Box(pca(p, c))$ is "true" iff every source of testimonial warrant in a community Σ "says" (or supports) that principal p is assigned to category c (i.e., $pca(p, c)$); $\Diamond(pca(p, c))$ is "true" iff some source of testimonial warrant in Σ "says" $pca(p, c)$; $M(pca(p, c))$ is "true" iff the majority of sources of testimonial warrant in Σ "say" $pca(p, c)$.

In addition, we introduce into our language a particular operator @ for external queries over remote knowledge bases such that φ @ ω intuitively reads as: "At remote source ω, φ is true", where φ can be either a literal or an aggregate function.

For the specification of the access control policies that we will introduce, we use standard ASP syntax with aggregate functions (in the sense of [2]). An *aggregate function* has the form $f(S)$ where S is a set, and f is a function name from the set of function names $\{\#count, \#sum, \#max, \#min, \#times\}$.

Definition 1 (Access Control Policy). *An access control policy is a set of ASP rules of the form $h \leftarrow b_1, \ldots, b_n$ where, h is a literal L or a counting operator applied to an instance of $\bigcirc pca(\_, \_)$ with $\bigcirc \in \{\Box, \Diamond, M\}$ and; $b_i := (not)L \mid \bigcirc pca(\_, \_) \mid L @ \omega \mid L_g \prec_1 f(S) \prec_2 R_g \mid L_g \prec_1 f(S) @ \omega \prec_2 R_g$;*

We recall that $L_g \prec_1 f(S) \prec_2 R_g$ is an *aggregate atom* where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$; and L_g and R_g (called *left guard*, and *right guard*, respectively) are terms.

As is conventional, variables in rules appear in the upper case; constants are in the lower case. We also restrict attention to policies that are defined as a finite set of safe (locally) stratified clauses. This means that our access control policies have a unique answer set.

We use aggregates in our approach and comparison operators on the numbers of sources of testimonial warrant to allow acceptors to specify flexibly the required degrees of testimonial support for access. We adopt features of the DLV language [3] for our purposes.

3 Implementation

In this section we present `secommunity`, an implementation of our framework in the DLV system. We first show how to represent, using DLV syntax, the operators presented throughout the paper. We then give a brief description of our system and we discuss some performance measures for it. In what follows, DLV code fragments are presented using `monospace` font. Henceforth, we view acceptors, oracles and contributors as DLV knowledge bases (KBs).

The implementation of our modal operators makes use of counting functions, to wit:

```
box(pca(P,C)) :- pcas(_,P,C)), #count{X : pcas(X,P,C), source(X)} = R,
                 #count{Y : source(Y)} = R.
diamond(pca(P,C)) :- pcas(X,P,C), source(X).
majority(pca(P,C)) :- pcas(_,P,C), #count{X : pcas(X,P,C), source(X)} = R1,
                      #count{Y : source(Y)} = R2, R3 = R2/2, R1 > R3.
```

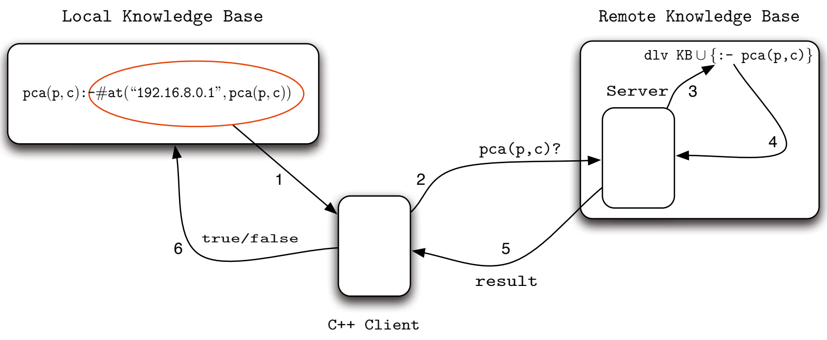


Fig. 1. Evaluation of #at

Notice that the definition of modal operators is based on *pcas* relationships issued by contributor sources of testimonial warrant.

3.1 Querying External Sources

To represent the distributed aspect of our approach, we defined in DLV-Complex[1] two external predicates (also called built-ins):

- #at(IP, G) which reads as: "At the remote DLV source indexed by IP, G holds."
- #f_at(IP, {P : Conj}, PX), which has several readings depending on the aggregate function f:
  - If f = count, then PX is unified with the value V, the sum of the elements in {P : Conj} at the remote DLV source indexed by IP, i.e., #count{P : Conj}=V.
  - If f = sum, then PX is unified with the value V representing the sum of numbers in the set {P : Conj} at the remote DLV source indexed by IP, i.e., #sum{P : Conj} = V.
  - If f = times, then PX is unified with the value V representing the product of numbers in the set {P : Conj} at the remote DLV source indexed by IP, i.e., #times{P : Conj} = V.
  - If f = min (resp. f = max), then PX is unified with the value V representing the minimum (resp. maximum) element in the set {P : Conj}, i.e, #max{P : Conj} = V.

For each external predicate (e.g., #at, #count_at) we associated a C++ program that queries the remote knowledge base indexed by IP. In Figure 1, we illustrate the execution cycle of the evaluation of #at("192.16.8.0.1", pca(p, c)) which can be resumed as follows:

- (1) When DLV has to evaluate whether #at("192.16.8.0.1", pca(p, c)) holds or not it performs a call to an external C++ function called client.
- (2) The server associated with the remote source is then contacted by the client, which asks whether pca(p, c) is in the answer set of the remote KB.

[1] https://www.mat.unical.it/dlv-complex

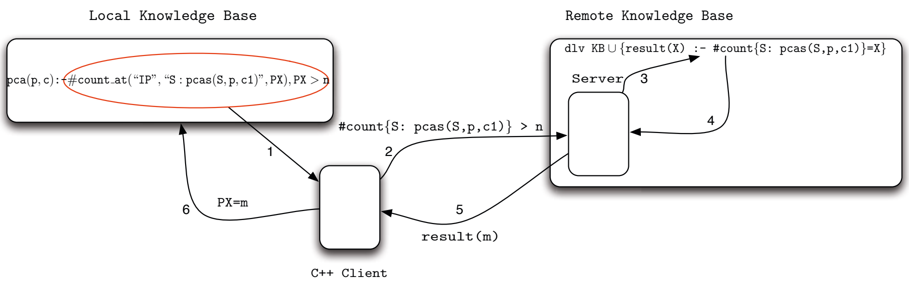

Local Knowledge Base
pca(p, c) :- #count_at("IP", "S : pcas(S, p, c1)", PX), PX > n
Remote Knowledge Base
dlv KB ∪ {result(X) :- #count{S: pcas(S,p,c1)}=X}
Server
C++ Client
#count{S: pcas(S,p,c1)} > n
result(m)
PX=m
1
2
3
4
5
6

Fig. 2. Evaluation of #count

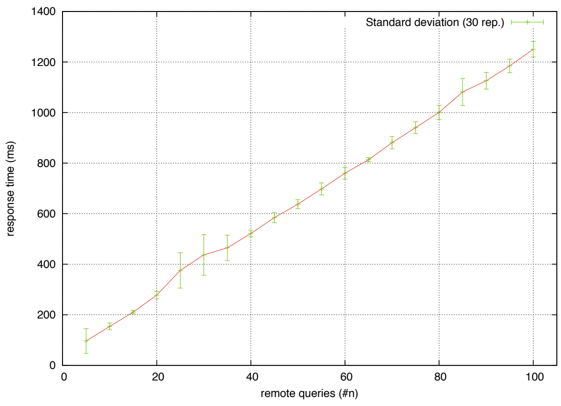


Fig. 3. Scalability of External Predicates

- (3) The server receives the request and checks locally whether the KB extended with constraint $:-$ pca(p, c) is contradictory , which means that pca(p, c) is in the answer set of KB.
- (4,5,6) The corresponding result is then forwarded back to the client which assigns true to #at("192.16.8.0.1", pca(p, c)) iff pca(p, c) holds at remote KB.

In Figure 2 we illustrate the execution cycle of the evaluation of

$$\#\mathtt{count\_at}(\text{“}\mathtt{IP}\text{”}, \text{“}\mathtt{S : pcas(S, p, c1)}\text{”}, \mathtt{PX}), \mathtt{PX} > \mathtt{n}$$

The steps are similar to those in Figure 1 with the exception that when DLV has to evaluate whether #count_at("IP", "S : pcas(S, p, c1)", PX), PX > n holds or not it has to query the remote KB (located at IP) to know the number of elements in the set {S : pcas(S, p, c1)} and then it must check whether this number is greater than n.

A working implementation (for Linux and Windows platforms) of secommunity with source code, documentation and examples is available at

http://www.di.unito.it/~genovese/tools/secommunity/secommunity.html

3.2 Tests and Results

In order to assess the overhead caused by the evaluation of external predicates, we studied the relationship between execution time and the number of external predicates necessary to evaluate a given rule. In particular, we consider the evaluation of rules of type

$$(T_n) \quad \mathtt{pca(p,c)} \;\mathtt{:\text{-}}\; \#\mathtt{at(IP,pca(p,c))}_1, \ldots, \#\mathtt{at(IP,pca(p,c))}_\mathtt{n}$$

In Figure 3 we report the main results of our experiments[2]. We plot the average and standard deviation of the time (in milliseconds) needed to evaluate 30 runs of rules $T_{(i \cdot 5)}$ with $1 \leq i \leq 20$. We note that the time to compute in DLV-Complex T_n grows *linearly* in n, while the standard deviation is due to network latency in the server's responses.

The performance measures offer some evidence to suggest that our approach is scalable for realistic applications.

4 Conclusions and Further Work

We have argued for a community-based approach for testimonial warrant in deciding access requests. We briefly described a policy specification language that include operators for describing unanimity of view in a community (via $\Box$), the existence of some contributor asserting ϕ ($\Diamond\phi$) and a majority operator M (which can be variously interpreted). We described how the framework that we defined can be represented in logic programming terms and we described an ASP-based implementation of our approach. We also presented some performance measures for an implementation of our approach that offer evidence of its scalability. We also note that although our focus has been on describing an approach for distributed access control, our proposal may be used in a variety of different application scenarios.

In this paper, we have been concerned with describing the implementation details for the secommunity framework that we have briefly outlined. In future work, we intend to develop the access control material that has not been our focus in this paper. On that, we propose to consider community membership issues (e.g., how to address the effects of changes to the community) and a richer delegation framework. Another matter for further work is to consider community disagreements. Oracles may introspectively reflect on the knowledge they possess with a view to resolving "conflicts" or "disagreements" within a community of contributors. Thus far, we have only considered a "democratic" form of community in which every contributor's assertions have the same weight. Accommodating variable measures of authority in terms of the assertions that contributors make to the community view is another issue that requires further investigation. The implications of accommodating these extensions in a logic programming context and in nonmonotonic policy specifications will also be matters for future work.

[2] We conduct our tests on the WAN of University of Torino, the client is a Mac Book Pro with a Intel Core 2 Duo, 4 GB of RAM running Mac OS X 10.6.5. As a server we used a Linux workstation with AMD Athlon, 2 GB of RAM running Ubuntu 10.4.

Acknowledgements. Valerio Genovese is supported by the National Research Fund, Luxembourg. The authors thank the reviewers for their comments, which proved to be helpful for improving the clarity of the paper.

References

1. Barker, S.: The next 700 access control models or a unifying meta-model? In: Proceedings of 14th ACM Symposium on Access Control Models and Technologies, SACMAT, pp. 187–196 (2009)
2. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI, pp. 847–852 (2003)
3. Leone, N., Faber, W.: The DLV project: A tour from theory and research to applications and market. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 53–68. Springer, Heidelberg (2008)
4. Li, N., Feigenbaum, J.: Nonmonotonicity, user interfaces, and risk assessment in certificate revocation. In: Syverson, P.F. (ed.) FC 2001. LNCS, vol. 2339, pp. 157–168. Springer, Heidelberg (2002)
5. Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust-management framework. In: Proceedings of 23rd IEEE Symposium on Security and Privacy, pp. 114–130 (2002)
6. Rivest, R.L.: Can we eliminate certificate revocation lists? In: Hirschfeld, R. (ed.) FC 1998. LNCS, vol. 1465, pp. 178–183. Springer, Heidelberg (1998)

Itemset Mining as a Challenge Application for Answer Set Enumeration*

Matti Järvisalo

Department of Computer Science, University of Helsinki, Finland

Abstract. We present an initial exploration into the possibilities of applying current state-of-the-art answer set programming (ASP) tools—esp. conflict-driven answer set enumeration—for mining itemsets in 0-1 data. We evaluate a simple ASP-based approach experimentally and compare it to a recently proposed framework exploiting constraint programming (CP) solvers for itemset mining.

1 Introduction

Answer set programming (ASP) has become a viable approach to solving various hard combinatorial problems. This success is based on the combination of an expressive modeling language allowing high-level declarations and optimized black-box solver technology following the success story of Boolean satisfiability (SAT) solving.

A somewhat specific feature of typical answer set solvers is support for enumerating all solutions (answer sets) of answer set programs. In this work we study an exciting novel application domain for answer set enumeration, namely, the data mining task of finding all *frequent itemset* from 0-1 data [1]. We show that itemset mining problems allow for simple and natural encodings as ASP with the help of the rich modeling language. Notably, typical itemset mining algorithm are somewhat problem specific, varying on the constraints imposed on itemsets. Surprisingly, constraint satisfaction techniques have only very recently been applied to itemset mining tasks [2,3]. In addition to the availability of black-box constraint solvers such as answer set enumerators, the additional benefit of constraint solvers is that the modeling languages enable solving novel itemset mining tasks by combining different itemset constraints in a natural way without having to devise new solving algorithms for specific mining tasks.

In this short paper, focusing on the standard [1] and maximal [4] frequent itemset mining problems, we evaluate the effectiveness of answer set enumeration as an itemset mining tool using a recent conflict-driven answer set enumeration algorithm [5], and compare this ASP approach to a recent approach based on generic constraint programming (CP) [2,3]. The results show that, even with simple encodings, ASP can be a realistic approach to itemset mining, and, on the other hand, that itemset mining is a well-motivated benchmark domain for answer set enumeration (adding to the currently relative few realistic applications of answer set enumeration).

* Research financially supported by Academy of Finland under grant 132812.

2 Itemset Mining

Assume a set $\mathcal{I} = \{1, ..., m\}$ of *items* and a set $\mathcal{T} = \{1, ..., n\}$ of *transactions*. Intuitively, a transaction $t \in \mathcal{T}$ consists of a subset of items from $\mathcal{I}$. An itemset database $\mathcal{D} \in \{0,1\}^{n \times m}$ is a binary matrix of size $n \times m$ that represents a set of transactions. Each row $\mathcal{D}_t$ of $\mathcal{D}$ represents a transaction t that consists of the set of items $\{i \in \mathcal{I} \mid \mathcal{D}_{ti} = 1\}$, where $\mathcal{D}_{ti}$ denotes the value on the ith column and tth row of $\mathcal{D}$.

The subsets of $\mathcal{I}$ are called *itemsets*. In itemset mining we are interested in finding itemsets that satisfy pre-defined constraints relative to an itemset database $\mathcal{D}$. Let $\varphi : 2^{\mathcal{I}} \rightarrow 2^{\mathcal{T}}$ be a function that maps an itemset $I \subseteq \mathcal{I}$ to the set $T \subseteq \mathcal{T}$ of transaction in which all its items occur, that is, $\varphi(I) = \{t \in T \mid \forall i \in I : \mathcal{D}_{ti} = 1\}$. The dual of φ is the function $\psi : 2^{\mathcal{T}} \rightarrow 2^{\mathcal{I}}$ that maps a set of transactions $T \subseteq \mathcal{T}$ to the set of all items from I included in all transactions in T, that is, $\psi(T) = \{i \in I \mid \forall t \in T : \mathcal{D}_{ti} = 1\}$.

Standard Frequent Itemsets. Assume a transaction database $\mathcal{D}$ over the sets $\mathcal{T}$ of transactions and $\mathcal{I}$ of items, and additionally a *frequency threshold* $\theta \in \{0, \ldots, |\mathcal{T}|\}$. Then the (traditional) frequent itemset problem [1] consists of finding the solution pairs (I, T), where $I \subseteq \mathcal{I}$ and $T \subseteq \mathcal{T}$, such that

$$T = \varphi(I) \tag{1}$$

$$|T| \geq \theta. \tag{2}$$

The first constraint requires that T must include all transactions in $\mathcal{D}$ that include all items in I. The second constraint requires that I is a frequent itemset, that is, the number of transactions in $\mathcal{D}$ in which all items in I occur must be at least θ. Notice that the second (*minimum frequency*) constraint is an anti-monotonic one: any subset of a frequent itemset is also a frequent itemset relative to a given threshold θ.

Various refinements of the traditional frequent itemset problem have been proposed. In addition to the traditional version, in this paper we consider the problem of finding *maximal* frequent itemsets [4], that is, frequent itemsets that are superset-maximal among the frequent itemsets of a given transaction database.

Maximal Frequent Itemsets. In addition to the constraints (1) and (2), the *maximality* constraint imposed in the maximal frequent itemset problem is

$$|\varphi(I')| < \theta \quad \forall I' \supset I, \tag{3}$$

that is, all supersets of a maximal frequent itemset are infrequent. Maximal frequent itemsets are a condensed representation for the set of frequent itemsets, constituting a border in the subset lattice of $\mathcal{I}$ between frequent and infrequent itemsets.

3 Itemsets as Answer Sets

We now consider two simple encodings of the standard and maximal frequent itemset problems as answer set programs. Due to the page limit we do not review details of the answer set semantics or the language used for expressing answer set programs. For more details on the input language, we refer the reader to the user's guide [6] of the Potassco bundle that includes the answer set enumerator we apply in the experiments.

```
1. item(I) :- db(_,I).
2. transaction(T) :- db(T,_).
3. { in_itemset(I) } :- item(I).
4. in_support(T) :- { conflict_at(T,I) : item(I) } 0, transaction(T).
5. conflict_at(T,I) :- not db(T,I), in_itemset(I), transaction(T).
6. :- { in_support(T) } N-1, threshold(N).
```

Fig. 1. The ASP(1) encoding of standard frequent itemset mining

```
1. item(I) :- db(_,I).
2. transaction(T) :- db(T,_).
3. { in_itemset(I) } :- item(I), N { in_support(T) : db(T,I) }, threshold(N).
4. in_support(T) :- { conflict_at(T,I) : item(I) } 0, transaction(T).
5. conflict_at(T,I) :- not db(T,I), in_itemset(I), transaction(T).
```

Fig. 2. The ASP(2) encoding of standard frequent itemset mining

We will intuitively explain the considered encoding referred to as ASP(1) (see Fig. 1) and ASP(2) (see Fig. 2). For both of the encodings, each answer set corresponds to a unique solution (I, T) of the itemset mining problem. Notice that there is an answer set for any dataset $\mathcal{D}$ and any threshold value θ, since by definition the empty set $\emptyset$ is always a frequent itemset. Although the encodings are quite similar, experiments show that the behavior of a state-of-the-art answer set enumerator varies notably depending of which encoding is used.

For presenting the transaction database $\mathcal{D}$, we use the predicate `db/2` and introduce the fact `db(t,i)` if and only if $\mathcal{D}_{ti} = 1$. The threshold θ is encoded using the predicate `threshold/1` by introducing the fact `threshold`(θ). The predicate `in_itemset/1` is true for an item i if and only if i is included in a frequent itemset I, encoding the most important part of a solution (I, T). The predicate `in_support/1` is true for a transaction t if and only if $t \in T$. Here the intuition is that, according to Eq. 1, each $t \in T$ has to *support* each $i \in I$ in the sense that t must include i (that is, $\mathcal{D}_{ti} = 1$). Additionally, we use the auxiliary predicates `item/1` (true for each item in $\mathcal{D}$), `transaction/1` (true for each transaction in $\mathcal{D}$), and `in_conflict/2`. The predicate `in_conflict/2`(t, i) is true for (t, i) if and only if transaction t does not support item i, that is, we have the *conflict* $\mathcal{D}_{ti} = 0$ and $i \in I$, violating Eq. 1.

Standard Frequent Itemset Mining. First consider the case of standard frequent itemset mining. Lines 1-2 in ASP(1) and ASP(2) are the same, simply stating that if $\mathcal{D}_{ti} = 1$ for some t, then i is an item (line 1), and similarly for transactions (line 2). The fact that a transaction t supports an itemset is also encoded in the same fashion in ASP(1) and ASP(2) on lines 4-5. Transaction t is in the support only if there is no conflict between t and the items in the itemset, that is, the number of true `conflict_at(`t, i`)`'s is zero (line 4, using a *cardinality constraint*). The `conflict_at/2` predicate is then defined on line 5: there is a conflict if $\mathcal{D}_{ti} = 0$ where i is in the frequent itemset.

The ASP(1) and ASP(2) encodings differ in how inclusion of items in the frequent itemset is represented. In ASP(1), on line 3 we "guess" for each item whether it is in the frequent itemset ({ `in_itemset(`i`)` } is the so called *choice* atom that is true regardless of whether `in_itemset(`i`)` is true). Given any choice of included items, the *integrity constraint* of line 6 requires that the number of transactions supporting the chosen itemset cannot be less than the frequency threshold, in accordance with the minimum frequency constraint (Eq. 2).

In ASP(2), we apply a more "direct" way of encoding inclusion of items in frequent itemsets (line 3): there is the choice of including an item if the particular item has enough supporting transactions (that is, at least as many as required by the threshold θ).

Maximal Frequent Itemset Mining. Based on the encodings for standard frequent itemset mining, including the additional maximality criterion for frequent itemsets requires only a small modification to both ASP(1) and ASP(2). Namely, for ASP(1) we *add* the rule `in_itemset(I) :- item(I), N { in_support(T) : db(T,I) }, threshold(N).` enforcing that any item that has sufficient support for inclusion in the frequent itemset has to be included. In contrast, for ASP(2) we *replace* the rule on line 3 with this same rule, in essence removing the *choice* from the original rule.

4 Experiments

Here we report on preliminary experiments addressing the efficiency of a state-of-the-art answer set enumerator on the ASP(1) and ASP(2) encodings of real-world datasets. As the answer set enumerator, we use the conflict-driven solver Clingo [5] (version 3.0.3, based on the Clasp ASP solver version 1.3.5)[1] with default settings. We also compare the performance of Clingo on ASP(1) and ASP(2) to that of FIM_CP[2] version 2.1 (using Gecode `http://www.gecode.org/` version 3.2.2), which is a recently proposed tool for itemset mining based on constraint programming [2,3]. The experiments were conducted under Ubuntu Linux on a 3.16-GHz Intel CORE 2 Duo E8500 CPU using a single core and 4-GB RAM. As benchmarks we used the preprocessed UCI datasets available at `http://dtai.cs.kuleuven.be/CP4IM/datasets/`, as used in evaluating FIM_CP [2,3]. Key properties of representative datasets, as supplied at this website, are shown in Table 1. We ran each solver for threshold θ values $0.95, 0.90, \ldots, 0.10, 0.08, \ldots, 0.02, 0.01$ times $|\mathcal{I}|$ for each dataset $\mathcal{D}$ until we observed the first timeout for a particular solver.

Table 1. Properties of the representative datasets

Dataset $\mathcal{D}$	transactions	items	density (%)	itemsets at $\theta = 0.1 \cdot \lvert\mathcal{I}\rvert$		grounding time (s)	
				standard	maximal	ASP(1)	ASP(2)
anneal	812	93	45	$> 147\,000\,000$	15 977	< 0.3	< 0.3
australian-credit	653	125	41	$> 165\,000\,000$	2 580 684	< 0.3	< 0.3
lymph	148	68	40	9 967 402	5191	< 0.1	< 0.1
mushroom	8124	119	18	155 734	453	< 3.4	< 2.5
splice-1	3190	267	21	1606	988	< 3.8	< 2.8

[1] `http://potassco.sourceforge.net/`. The options `-n 0 -q` were used for computing all solutions and suppressing printing of solutions.

[2] `http://dtai.cs.kuleuven.be/CP4IM/`. The option `-output none` was used for suppressing printing of solutions.

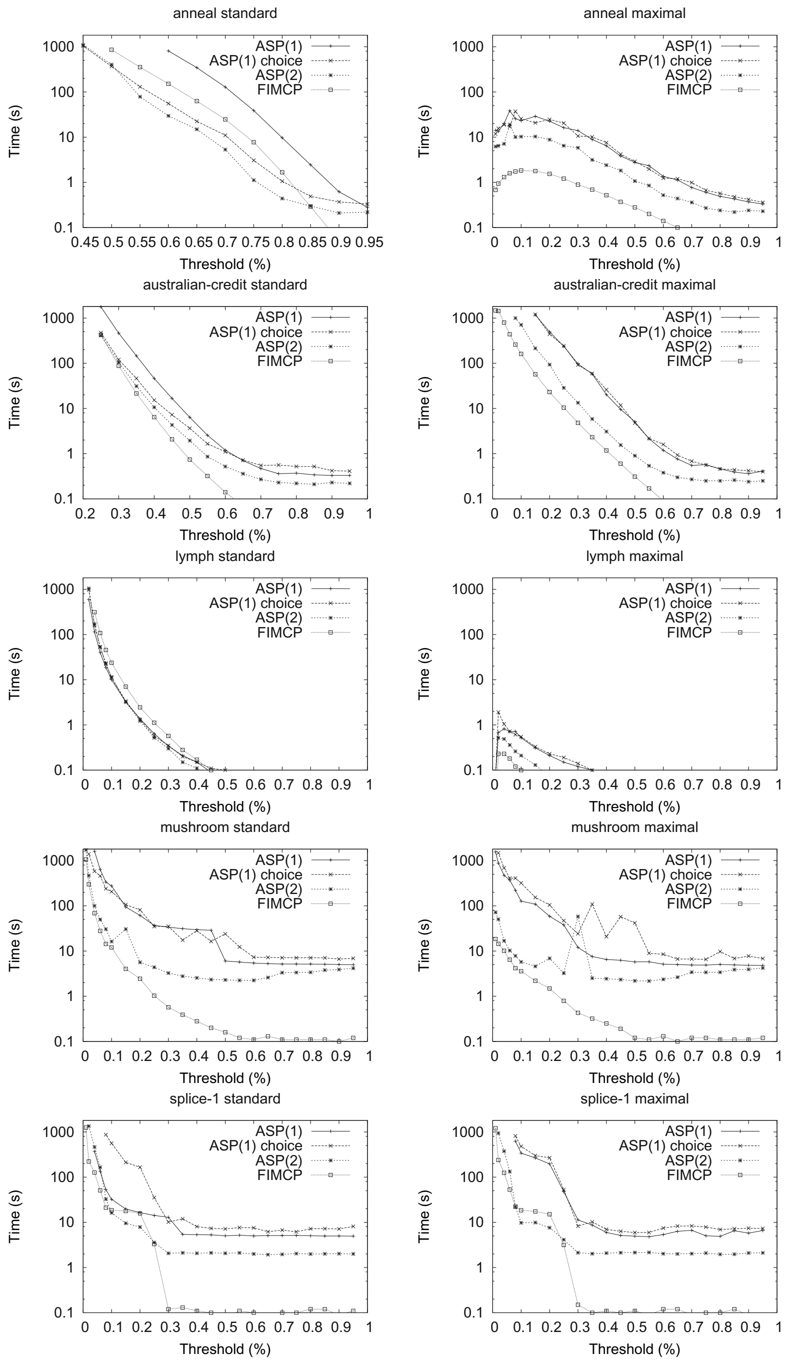


Fig. 3. Comparison of the CP and ASP approaches to standard and maximal frequent itemset mining on representative datasets

Results for a representative set of benchmarks are shown in Fig. 3, with observed upper bounds on the times used for grounding shown in Table 1. Grounding time is included in the plots. For the standard frequent itemset problem (left column), we observe that the ASP(2) encoding is almost always better than ASP(1). For the most dense dataset *anneal* (recall Table 1 – here density is defined as the percentage of 1's in $\mathcal{D}$) we observe that ASP(2) is the most effective one, being multiple times more effective than the FIM_CP approach. Also for the other two relatively dense datasets, ASP(2) is either slightly better than (on *lymph*) or approaches the performance (on *australian-credit*) of FIM_CP. This is an intriguing observation, since dense datasets can be considered harder to mine because of the large number of candidate itemsets. For the remaining two datasets, we observe that the performance of ASP(2) approaches that of FIM_CP, even being more effective at low threshold values on *splice-1*.

For the maximal itemset problem (right column) we observe that FIM_CP is the most efficient one, with the exception that for the *splice-1* dataset, the ASP(2) encoding dominates at the more difficult threshold values $\leq 0.20 \cdot |\mathcal{I}|$.

We also conducted a preliminary experiment on the effect of *decomposing* the cardinality and choice constructs in ASP(1) and ASP(2) using the build-in decompositions of Clingo. This is motivated by evidence of varied applications of constraint satisfaction tools in which decomposing complex constraints into lower level entities has resulted in improved performance. In this case, decomposing cardinalities seemed to generally degrade performance. However, decomposing only the choice constructs (using `--trans-ext=choice` in Clingo) in the standard frequent itemset encodings gave interesting results; see "ASP(1) choice" in Fig. 3. Namely, the performance of ASP(1) on *anneal* became even better than that of FIM_CP, but degraded further on *splice-1*. For ASP(2) we observed no notable differences.

Finally, we noticed that Smodels (with and without lookahead) is very ineffective on these problems compared to Clasp, and hence we excluded the Smodels data from the plots for clarity. However, in-depth experiments with other solution enumerating solvers (including, e.g., DLV) remains as future work, in addition to experimenting with different search heuristics and other search space traversal options offered by Clasp.

5 Conclusions

We propose itemset mining as a novel application and benchmark domain for answer set enumeration. The behavior of two simple ASP encodings varies depending on whether maximality of itemsets is required; the behavior of the "better" encoding can exceed that of a recent CP-based approach. We also observed that even small changes in the encoding—including decompositions—can reflect in notable performance differences when enumerating all solutions. This motivates further work on more effective encodings and on the interplay between answer set enumeration search techniques and modelling, with the possibility of optimizing solver heuristics towards data mining tasks. Additional current work includes finding dataset properties that imply good performance of the ASP approach, and encodings of other data mining tasks as ASP.

References

1. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast discovery of association rules. In: Advances in Knowledge Discovery and Data Mining, pp. 307–328. AAAI Press, Menlo Park (1996)
2. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proc. KDD, pp. 204–212. ACM, New York (2008)
3. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for data mining and machine learning. In: Proc. AAAI. AAAI Press, Menlo Park (2010)
4. Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., Yiu, T.: MAFIA: A maximal frequent itemset algorithm. IEEE Trans. Knowl. Data Eng. 17(11), 1490–1504 (2005)
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 136–148. Springer, Heidelberg (2007)
6. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo` (2008), `http://potassco.sourceforge.net/`

Causal Reasoning for Planning and Coordination of Multiple Housekeeping Robots

Erdi Aker[1], Ahmetcan Erdogan[2], Esra Erdem[1], and Volkan Patoglu[2]

[1] Computer Science and Engineering, Faculty of Engineering and Natural Sciences
Sabancı University, İstanbul, Turkey

[2] Mechatronics Engineering, Faculty of Engineering and Natural Sciences
Sabancı University, İstanbul, Turkey

Abstract. We consider a housekeeping domain with multiple cleaning robots and represent it in the action language $\mathcal{C}+$. With such a formalization of the domain, a plan can be computed using the causal reasoner CCALC for each robot to tidy some part of the house. However, to find a plan that characterizes a feasible trajectory that does not collide with obstacles, we need to consider geometric reasoning as well. For that, we embed motion planning in the domain description using external predicates. For safe execution of feasible plans, we introduce a planning and monitoring algorithm so that the robots can recover from plan execution failures due to heavy objects that cannot be lifted alone. The coordination of robots to help each other is considered for such a recovery. We illustrate the applicability of this algorithm with a simulation of a housekeeping domain.

1 Introduction

Consider a house consisting of three rooms: a bedroom, a living room and a kitchen as shown in Fig. 1. There are three cleaning robots in the house. The furniture is stationary and their locations are known to the robots a priori. Other objects are movable. There are three types of movable objects: books (green pentagon shaped objects), pillows (red triangular objects) and dishes (blue circular objects). Some objects are heavy and cannot be moved by one robot only; but the robots do not know which movable objects are heavy. The goal is for the cleaning robots to tidy the house collaboratively in a given number of steps. This domain is challenging from various aspects:

- It requires representation of some commonsense knowledge. For instance, in a tidy house, books are in the bookcase, dirty dishes are in the dishwasher, pillows are in the closet. In that sense, books are expected to be in the living room, dishes in the kitchen and pillows in the bedroom. Representing such commonsense knowledge and integrating it with the action domain description (and the reasoner) is challenging.
- A robot is allowed to be at the same location with a movable object only if the object is being manipulated (attached, detached or carried); otherwise, robot-robot, robot-stationary object and robot-moveable object collisions are not permitted. Due to these constraints, representing preconditions of (discrete) actions that require (continuous) geometric reasoning for a collision-free execution is challenging.

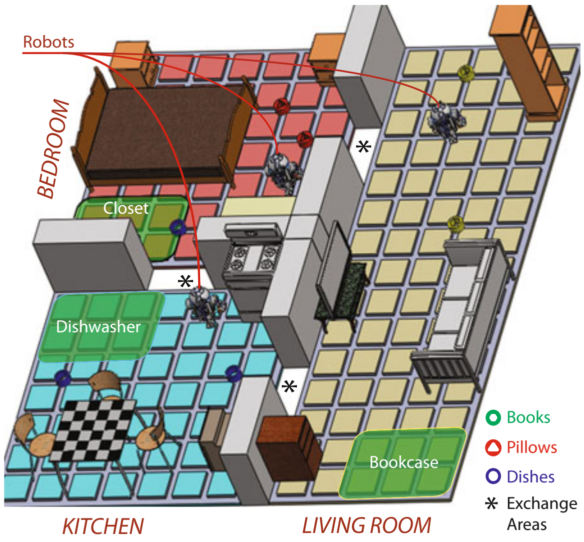


Fig. 1. Simulation environment for housekeeping domain

For instance, moving to some part of a room may not be possible for a robot because, although the goal position is clear, it is blocked by a table and a chair and the passage between the table and the chair is too narrow for the robot to pass through.
– Solving the whole housekeeping problem may not be possible because the formalization gets too large for the reasoner. In that case, we can partition the housekeeping problem into smaller parts (e.g., each robot can tidy a room of the house). However, then the robots must communicate with each other to tidy the house collaboratively. For instance, if a robot cannot move a heavy object to its goal position, the robot may ask another robot for help. If the robot that cleans kitchen finds a book on the floor, then the robot should transfer it to the robot that cleans the living room, by putting the book in the exchange area between kitchen and living room. Coordination of the robots in such cases, subject to the condition that the house be tidied in a given number of steps, is challenging.

We handle these challenges by representing the housekeeping domain in the action description language $\mathcal{C}+$ [3] as a set of "causal laws" (Section 2) and using the causal reasoner CCALC [7] for planning (Section 3), like in [2], in the style of cognitive robotics [5]. For the first two challenges, we make use of external predicates. We represent commonsense knowledge as a logic program, and use the predicates defined in the logic program as external predicates in causal laws. Similarly, we can implement collision checks as a function in the programming language C++, and use these functions as external predicates in causal laws. For the third challenge, we introduce a planning and monitoring algorithm that solves the housekeeping problem by dividing it into smaller problems and then combining their solutions, that coordinates multiple cleaning robots for a common goal (Section 4).

2 Representation of Housekeeping Domain

We view the house as a grid. The robots and the endpoints of objects are located at grid-points. We consider the fluents `at(TH,X,Y)` ("thing `TH` is at `(X,Y)`") and `connected(R,EP)` ("robot `R` is connected to endpoint `EP`"). We also consider the actions `goto(R,X,Y)` ("robot `R` goes to `(X,Y)`"), `detach(R)` ("robot `R` detaches from the object it is connected to"), and `attach(R)` ("robot `R` attaches to an object").

Using these fluents and actions, the housekeeping domain is represented in $\mathcal{C}+$ as described in [1]. Let us describe briefly two aspects of this representation: embedding geometric reasoning in causal reasoning, and integrating commonsense knowledge in the action domain description.

Embedding geometric reasoning. CCALC allows us to include "external predicates" in causal laws. These predicates/functions are not part of the signature of the domain description (i.e., they are not declared as fluents or actions). They are implemented as functions in some programming language of the user's choice, such as C++. External predicates take as input not only some parameters from the action domain description (e.g., the locations of robots) but also detailed information that is not a part of the action domain description (e.g., geometric models). They are used to externally check some conditions under which the causal laws apply, or externally compute some value of a variable/fluent/action. For instance, suppose that the external predicate `path(R,X,Y,X1,Y1)`, implemented in C++ based on Rapidly exploring Random Trees (RRTs) [4], holds if there is a collision-free path between `(X,Y)` and `(X1,Y1)` for the robot `R`. Then we can express that the robot `R` cannot go from `(X1,Y1)` to `(X,Y)` where `path(R,X,Y,X1,Y1)` does not hold, by a causal law presented to CCALC:

```
nonexecutable goto(R,X,Y) if at(R,X1,Y1)
   where -path(R,X1,Y1,X,Y).
```

Integrating commonsense knowledge. To clean a house, the robots should have an understanding of the following: tidying a house means that the objects are at their desired locations. For that, first we declare a "statically determined fluent" describing that the endpoint of an object is at its expected position in the house, namely `at_desired_location(EP)`, and define it as follows:

```
caused at_desired_location(EP) if at(EP,X,Y)
   where in_place(EP,X,Y).
default -at_desired_location(EP).
```

The second causal law above expresses that normally the movable objects in an untidy house are not at their desired locations. The first causal law formalizes that the endpoint `EP` of an object is at its desired location if it is at some "appropriate" position `(X,Y)` in the right room. Here `in_place/3` is defined externally.

After defining `at_desired_location/1`, we can define `tidy` by a "macro":

```
:- macros tidy -> [/\EP | at_desired_location(EP)].
```

Finally, the robots need to know that books are expected to be in the bookcase, dirty dishes in the dishwasher, and pillows in the closet. Moreover, a bookcase is expected to be in the living-room, dishwasher in the kitchen, and the closet in the bedroom. We describe such background knowledge externally as a Prolog program. For instance, the external predicate `in _place/3` is defined as follows:

```
in_place(EP,X,Y) :- belongs(EP,Obj), type_of(Obj,Type),
   el(Type,Room), area(Room,Xmin,Xmax,Ymin,Ymax),
   X>=Xmin, X=<Xmax, Y>=Ymin, Y=<Ymax.
```

Here `belongs(EP,OBJ), type_of(OBJ,Type)` describes the type `Type` of an object `Obj` that the endpoint `EP` belongs to, and `el(Type,Room)` describes the expected room of an object of type `Type`. The rest of the body of the rule above checks that the endpoint's location `(X,Y)` is a desired part of the room `Room`.

3 Reasoning about Housekeeping Domain

Given the action domain description and the background and commonsense knowledge above, we can solve various reasoning tasks, such as planning, using CCALC. However, the overall planning problem for three cleaning robots may be too large (considering the size of the house, number of the objects, etc.). In such cases, we can divide the problem into three smaller planning problems, assigning each robot to tidy a room of the house in a given number of steps.

Consider the housekeeping domain described above: Robot 1 is expected to tidy the living room, Robot 2 the bedroom, and Robot 3 the kitchen. Suppose that the locations of the movable objects are known to the robots a priori. Robot 1 knows that there are two books, `comics1` and `novel1`, on the living room floor. Robot 2, on the other hand, knows that there are two pillows, `redpillow1` and `bluepillow1`, and a plate, `plate1`, on the bedroom floor. The robots also know where to collect the objects. For instance, Robot 2 knows that, in the bedroom, the closet occupies the rectangular area whose corners are at `(5,0), (5,3), (7,0), (7,3)`. Robot 2 also knows that the objects that do not belong to bedroom, such as `plate1` of type `dish`, should be deposited to the exchange area between bedroom and kitchen, that occupies the points `(3,7)-(5,7)`. Planning problems for each robot are shown in Table 1. For instance, in the living room, initially Robot 1 is at `(3,2)`, whereas the books `comics11` and `novel11` are located at `(1,2)` and `(6,3)`. The goal is to tidy the room and make

Table 1. Planning problems for each robot

	Robot 1 in Living Room	Robot 2 in Bedroom	Robot 3 in Kitchen
Initial State	`at(r1,3,2)` `at(novel1,6,3)` `at(comics1,1,2)`	`at(r2,5,6)` `at(bluepillow1,3,6)` `at(redpillow1,2,5)` `at(plate1,6,3)`	`at(r3,1,5)` `at(spoon1,3,6)` `at(pan1,3,1)`
Goal	`tidy, free`	`tidy, free`	`tidy, free`

sure that the robot is free (i.e., not attached to any objects). Here `free` is a macro, like `tidy`. Given these planning problems, CCALC computes a plan for each robot.

4 Monitoring the Cleaning Robots

Once a plan is computed for each robot by CCALC, each robot starts executing it. However, a plan execution may fail: while most of the moveable objects are carried with only one robot, some of these objects are heavy and their manipulation requires two robots; the robots do not know in advance which objects are heavy, but discover a heavy object only when they attempt to move it.

When a plan fails because a robot attempts to manipulate a heavy object, the robot asks for assistance from other robots so that the heavy object can be carried to its destination. However, in order not to disturb the other robots while they are occupied with their own responsibilities, the call for help is delayed as much as possible. With the observation that the manipulation of the heavy object takes 4 steps (get to the heavy object, attach to it, carry it, detach from it), this is accomplished by asking CCALC to find a new plan that manipulates the heavy object within the last $i = 4, 5, 6, ...$ steps of the plan only. If there is such a plan, one of the robots who are willing to help gets prepared (e.g., detaches from the object it is carrying, if there is any) and goes to the room of the robot who requests help. (Currently, task allocation is done randomly.) If no such plan is computed, then the robot does not delay asking for help; it calls for immediate

Table 2. Execution of the plans computed by CCALC for each robot. The rows that are not labeled by a time step are not part of these plans, but are implemented at the low-level.

Step	Robot 1 in Living Room	Robot 2 in Bedroom	Robot 3 in Kitchen
1	`goto(r1,6,3)`	`goto(r2,6,3)`	`goto(r3,3,1)`
2	`attach(r1,novel1)`	`attach(r2,plate1)`	`attach(pan1)` - FAILURE (Heavy object)
...	...	...	...
	Get ready to help r3		
7	help r3	`goto(r2,5,2)`	`goto(r3,3,1)` `goto(r1,4,1)`
8	help r3	`detach(r2)`	`attach(r3,pan1)` `attach(r1,pan2)`
9	help r3	`goto (r2,2,5)`	`goto(r3,0,1)` `goto(r1,1,1)`
10	help r3	`attach(r2,redpillow1)`	`detach(r3)` `detach(r1)`
	Get ready to continue plan		
11	`goto(r1,13,2)`	`goto(r2,7,1)`	-
12	`detach(r1)`	`detach(r2)`	-
...	...	...	...

help and waits for assistance to arrive. For that, the robot asks CCALC to compute a new plan that involves moving the heavy object to its goal position. After that, trajectories are computed for the robot itself and the helper robot; and these trajectories are followed concurrently.

Table 2 shows some parts of the execution of plans by Robots 1–3. Robot 1 executes Plan 1 and goes to kitchen at time step 7 to help Robot 3 to move a heavy object to its goal position. Robot 3 on the other hand starts executing a plan, but at time step 2, finds out that the pan `pan1` he wants to move is too heavy. Then Robot 3 goes to a safe state and asks for help to carry the heavy object to its goal position.

5 Conclusion

We formalized a housekeeping domain with multiple cleaning robots, in the action description language $\mathcal{C}+$, and solved some housekeeping problem instances using the reasoner CCALC as part of a planning and monitoring framework. While representing the domain, we made use of some utilities of CCALC: external predicates are used to embed geometric reasoning in causal laws. To represent commonsense knowledge and background knowledge, we made use of external predicates/functions and macros in causal laws. The extension of our approach to handle collaborations of heterogenous robots (as in [6]) is part of our ongoing work.

Acknowledgments

This work has been partially supported by Sabanci University IRP Grant.

References

1. Aker, E., Erdogan, A., Erdem, E., Patoglu, V.: Housekeeping with multiple autonomous robots: Representation, reasoning and execution. In: Proc. of Commonsense (2011)
2. Caldiran, O., Haspalamutgil, K., Ok, A., Palaz, C., Erdem, E., Patoglu, V.: Bridging the gap between high-level reasoning and low-level control. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 342–354. Springer, Heidelberg (2009)
3. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence 153, 49–104 (2004)
4. Lavalle, S.M.: Rapidly-exploring random trees: A new tool for path planning. Tech. Rep. (1998)
5. Levesque, H., Lakemeyer, G.: Cognitive robotics. In: Handbook of Knowledge Representation. Elsevier, Amsterdam (2007)
6. Lundh, R., Karlsson, L., Saffiotti, A.: Autonomous functional configuration of a network robot system. Robotics and Autonomous Systems 56(10), 819–830 (2008)
7. McCain, N., Turner, H.: Causal theories of action and change. In: Proc. of AAAI/IAAI, pp. 460–465 (1997)

ASPIDE: Integrated Development Environment for Answer Set Programming

Onofrio Febbraro[1], Kristian Reale[2], and Francesco Ricca[2]

[1] DLVSystem s.r.l. - P.zza Vermicelli, Polo Tecnologico, 87036 Rende, Italy
febbraro@dlvsystem.com
[2] Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{reale,ricca}@mat.unical.it

Abstract. Answer Set Programming (ASP) is a truly-declarative programming paradigm proposed in the area of non-monotonic reasoning and logic programming. In the last few years, several tools for ASP-program development have been proposed, including (more or less advanced) editors and debuggers. However, ASP still lacks an Integrated Development Environment (IDE) supporting the entire life-cycle of ASP development, from (assisted) programs editing to application deployment. In this paper we present *ASPIDE*, a comprehensive IDE for ASP, integrating a cutting-edge *editing tool* (featuring dynamic syntax highlighting, on-line syntax correction, autocompletion, code-templates, quick-fixes, refactoring, etc.) with a collection of user-friendly *graphical tools* for program composition, debugging, profiling, database access, solver execution configuration and output-handling.

1 Introduction

Answer Set Programming (ASP) [1] is a declarative programming paradigm which has been proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find such a solution [2]. The language of ASP is very expressive [3]. Furthermore, the availability of some efficient ASP systems [4,5,6,7,8,9,10,11,12,13,14,15] made ASP a powerful tool for developing advanced applications. ASP applications belong to several fields, from Artificial Intelligence [14,16,17,18,19,20,21] to Information Integration [22], and Knowledge Management [23,24,25]. These applications of ASP have confirmed, on the one hand, the viability of the exploitation in real application settings and, very recently, stimulated some interest also in industry [26]. On the other hand, they have evidenced the lack of effective development environments capable of supporting the programmers in managing large and complex projects [27]. It is nowadays recognized [27] that this may discourage the usage of the ASP programming paradigm, even if it could provide the needed reasoning capabilities at a lower (implementation) price than traditional imperative languages. Note also that, the most diffused programming languages always come with the support of SDKs featuring a rich set of tools that significantly simplify both programming and maintenance tasks.

In order to facilitate the design of ASP applications, some tools for ASP-program development were proposed in the last few years (consider, for instance, the aim of SEA workshop series [28,29]), including editors [30,31] and debuggers [32,33,34,35,36]. However, ASP still lacks an Integrated Development Environment (IDE) supporting the entire life-cycle of ASP development, from (assisted) programs editing to application deployment.

This paper provides a contribution in this setting. In fact, it presents *ASPIDE*, a comprehensive IDE for ASP, which integrates a cutting-edge *editing tool* (featuring dynamic syntax highlighting, on-line syntax correction, autocompletion, code-templates, quick-fixes, refactoring, etc.) with a collection of user-friendly *graphical tools* for program composition, debugging, profiling, DBMS access, solver execution configuration and output-handling. Currently, the system is able to load and store ASP programs in the syntax of the ASP system DLV [4], and supports the `ASPCore` language profile employed in the ASP System Competition 2011[37]. Data base management is compliant with DLV^{DB} [38] language directives.

A comprehensive feature-wise comparison with existing environments for developing logic programs is also reported in this paper, which shows that *ASPIDE* represents a step forward in the present state of the art of tools for ASP programs development.

2 Main Features of *ASPIDE*

The main functionalities provided by *ASPIDE* are first summarized below, and then described in more detail. In particular, *ASPIDE* provides the following features:

- *Workspace management*. The system allows to organize ASP programs in projects à la Eclipse, which are collected in a special directory (called workspace).
- *Advanced text editor*. The editing of ASP files is simplified by an advanced text editor, which provides several functionalities: from simple text coloring to auto completion of predicates and variables names.
- *Outline navigation*. *ASPIDE* creates an outline view which graphically represents program elements.
- *Dependency graph*. The system provides a graphical representation of the dependency graph of a program.
- *Dynamic code checking and errors highlighting*. Syntax errors and relevant conditions (like safety) are checked *while typing programs*: portions of code containing errors or warnings are immediately highlighted.
- *Quick fix*. The system suggests quick fixes to reported errors or warnings, and applies them (on request) by automatically changing the affected part of code.
- *Dynamic code templates*. Writing of repeated programming patterns (like transitive closure or disjunctive rules for guessing the search space) is assisted by advanced auto-completion generating rules or part of them.
- *Test suite*. The user can define test suites on the line with the principle of unit testing diffused in the development of software with iterative languages.
- *Debugger and Profiler*. Semantic errors detection as well as code optimization can be done by exploiting graphic tools.

- *Configuration of the execution*. This feature allows to configure input programs and execution options.
- *Presentation of results*. The output of the program (either answer sets, or query results) are visualized in a tabular representation or in a text-based console.
- *Visual Editor*. The users can *draw* logic programs by exploiting a full graphical environment that offers a QBE-like tool for building logic rules. The user can switch, every time he needs, from the text editor to the visual one (and vice-versa) thanks to a reverse-rengineering mechanism from text to graphical format.
- *Interaction with databases*. Interaction with external databases is made easy by a fully graphical import/export tool that automatically generates mappings by following the DLV^{DB} Typ files specifications [38]. Text editing of Typ mappings is also assisted by syntax coloring and auto-completion.

In the following, we describe in more detail the above mentioned functionalities.

Workspace organization. The system allows for organizing ASP programs in projects à la Eclipse. This facilitates the development of complex applications by organizing modules (or projects) in a space where either different parts of an encoding or several equivalent encodings solving the same problem are stored. In particular, *ASPIDE* allows to manage: (i) *DLV files* containing ASP programs in both DLV syntax [4] and ASPCore [37]; (ii) *TYP files* specifying a mapping between program predicates and database tables in DLV^{DB} syntax; (iii) *TEST files* containing a set of directives conceived for defining unit tests for the created logic programs.

Advanced text editor. The presence of an editor that provides a set of advanced features is indispensable for a good development environment. In particular, besides the core functionality that basic text editors offer (like, code line numbering, find/replace, undo/redo, copy/paste, etc.), *ASPIDE* offers others advanced functionalities:

- *Text coloring*. *ASPIDE* exploits different colors for outlining key words (like " :– ") predicate names, variables, strings, comments etc. Predicates involved in database mappings are also specifically marked.
- *Automatic completion*. The system is able to complete (on request) predicate names, as well as variable names. Predicate names are both learned while writing, and extracted from the files belonging to the same project; variables are suggested by taking into account the rule we are currently writing. This helps while developing either an alternative encoding for the same problem (input/intermediate predicate names are ready to be suggested after the first file is completed) or when the same solution is divided in several files. When several possible alternatives for completion are available the system shows a pop-up dialog.
- *Refactoring*. The refactoring tool allows to modify in a guided way, among others, predicate names and variables. For instance, variable renaming in a rule is done by considering bindings of variables, so that common side effects of find/replace are avoided by ensuring that variables/predicates/strings occurring in other expressions remain unchanged.

Outline navigation. *ASPIDE* creates a graphic outline of both programs and Typ files, which graphically represents language statements. Regarding the programs, the outline is a tree representation of each rule where in the first level of the tree there are the head atoms, while the second level corresponds to the bodies. Each item in the outline can be used to quickly access the corresponding line of code (a very useful feature when dealing with long files), and also provides a graphical support for building rules in the graphical editor (see [39]).

Dependency graph. The system provides a graphical representation of the (non-ground) dependency graph associated to the project. Several variants of the dependency graph are supported depending on whether both positive and negative dependencies are considered. The graph of strongly connected components (playing an important role in the instantiation of the program) can be displayed also.

Dynamic code checking and errors highlighting. Programs are parsed while writing, and both errors or possible warnings are immediately outlined without the need of saving files. In particular, syntax errors as well as mismatching predicate arities and safety conditions are checked. Note that, the checker considers the entire project, and warns the user by indicating e.g., that atoms with the same predicate name have different arity in several files. This condition is usually revealed only when programs divided in multiple files are run together.

Quick fix. The system suggests quick fixes to reported errors or warnings, and applies them (on request) by automatically changing the affected part of code. This can be activated by clicking on the line of code which contains an error/warning and choosing from a popup window the desired fix among several suggestions, e.g., safety problems can be fixed by correcting variable names or by projecting out "unsafe" variables through an auxiliary rule (which will be automatically added).

Code template. *ASPIDE* provides support for automated writing of parts of rules (guessing patterns, aggregates, etc.), as well as automated writing of entire subprograms (e.g., transitive closure rules) by exploding code templates. Note that, templates can be also user defined by writing DLT [40] files.

Test suite. A very important phase in the development of software is testing. In *ASPIDE* we have defined a syntax for writing and running tests in the style of JUnit. One can write assertions regarding the number of answer sets or the presence/absence of an atom in the results of a test.

Debugger and Profiler. We have embedded in *ASPIDE* the debugging tool *spock* [32]. To this end we have worked on the adaptation of spock for dealing with the syntax of the DLV system. Moreover, we developed a graphical user interface that wraps the above mentioned tool. Regarding the profiler, we have fully embedded the graphical interface presented in [41].

Configuration of the execution. *ASPIDE* provides a form for configuring and managing solver execution. A run configuration allows to set the solver/system executable, setup invocation options and input files. Moreover, *ASPIDE* supports a graphical tool for composition of workflow executions (e.g., combining several solver/system calls), which are transparently compiled in perl scripts.

Presentation of the results. The ASP solvers output results in textual form. For making comfortable the browsing of results, *ASPIDE* visualizes answer sets by combining tabular representation of predicates with a comfortable tree-like representation for handling multiple answer sets. The result of the execution can be also saved in text files for subsequent analysis.

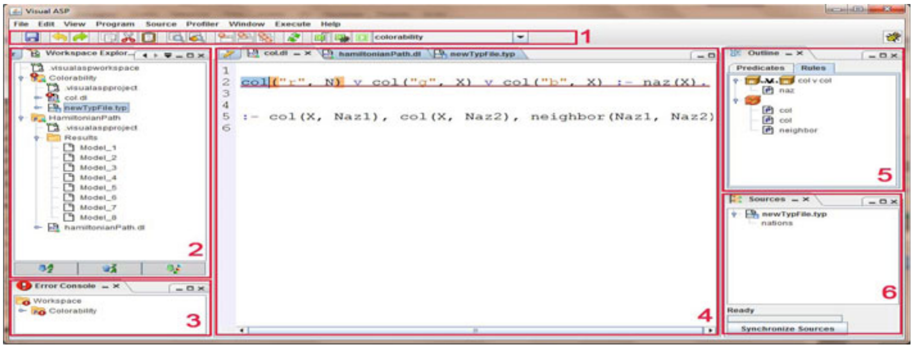
Visual editor. *ASPIDE* offers also the possibility to draw programs using a fully graphical environment by embedding and extending the *Visual Editor* tool which offers a QBE-like approach; it allows, for example, to create graphical bindings of variables using "joins" between predicates. For a detailed explanation on how the Visual Editor works, see [39]. An important feature offered by *ASPIDE* is *reverse engineering* that allows to switch between textual and visual representation of programs.

Interaction with databases. Interaction with external databases is useful in several applications (e.g., [22,26,19]). *ASPIDE* allows to access external databases by exploiting a graphical tool connecting to DBMSs via JDBC. Imported sources are emphasized also in the program editor by exploiting a specific color. Database oriented applications can be run by setting DLV^{DB} as solver in a run configuration. A data integration scenario [22] can be implemented by exploiting this feature.

3 Interface Overview and Implementation

The system interface of *ASPIDE* is depicted in Figure 1, where the main components are outlined in different numbered zones. In the upper part of the interface (zone 1) a toolbar allows the user to call the most common operations of the system (from left to right: save files, undo/redo, copy & paste, find & replace, switch between visual to text editor, run the solver/profiler/debugger). In the center of the interface there is the main editing area (zone 4), organized in a multi-tabbed panel possibly collecting several open files. The left part of the interface is dedicated to the explorer panel (zone 2), and to the error console (zone 3). The explorer panel lists projects and files included in the workspace, while the error console organizes errors and warnings according to the project and files where they are localized. On the right, there are the outline panel (zone 5) and the sources panel (zone 6). The first shows an outline of the currently edited file, while the latter reports a list of the database sources which might be mapped to some predicate of the current project. The one shown in Figure 1 is the standard appearance of the system, which can be however modified, since panels can be moved as the user likes. A comprehensive description of *ASPIDE* is available in the online manual published in the system web site `http://www.mat.unical.it/ricca/aspide`.

ASPIDE is written in Java by following the Model View Controller (MVC) pattern. A *core* module manages, by means of suitable data structures, projects, files content, system status (e.g., error lists, active connections to DBMSs etc.), and external component management (e.g., interaction with solver/debugger/profiler). Any update to the information managed by *ASPIDE* is obtained by invoking methods of the core, while, *view*



```
col("r", N) v col("g", X) v col("b", X) :- naz(X).

:- col(X, Naz1), col(X, Naz2), neighbor(Naz1, Naz2)
```

Fig. 1. The *ASPIDE* graphical user interface

modules (graphically implemented by interface panels) are notified by proper events in case of changes in the system status. *ASPIDE* exploits: (i) the JGraph (`http://www.jgraph.com/`) library in the visual editor, in the dependency graph and in the workflow modules; (ii) the DLV Wrapper [42] for interacting with DLV; and, (iii) JDBC libraries for database connectivity. Debugging and profiling are implemented by wrapping the tool *spock* [32], and the *DLV profiler* [41], respectively. Visual editing is handled by including an extended version of the *VisualASP* tool [39].

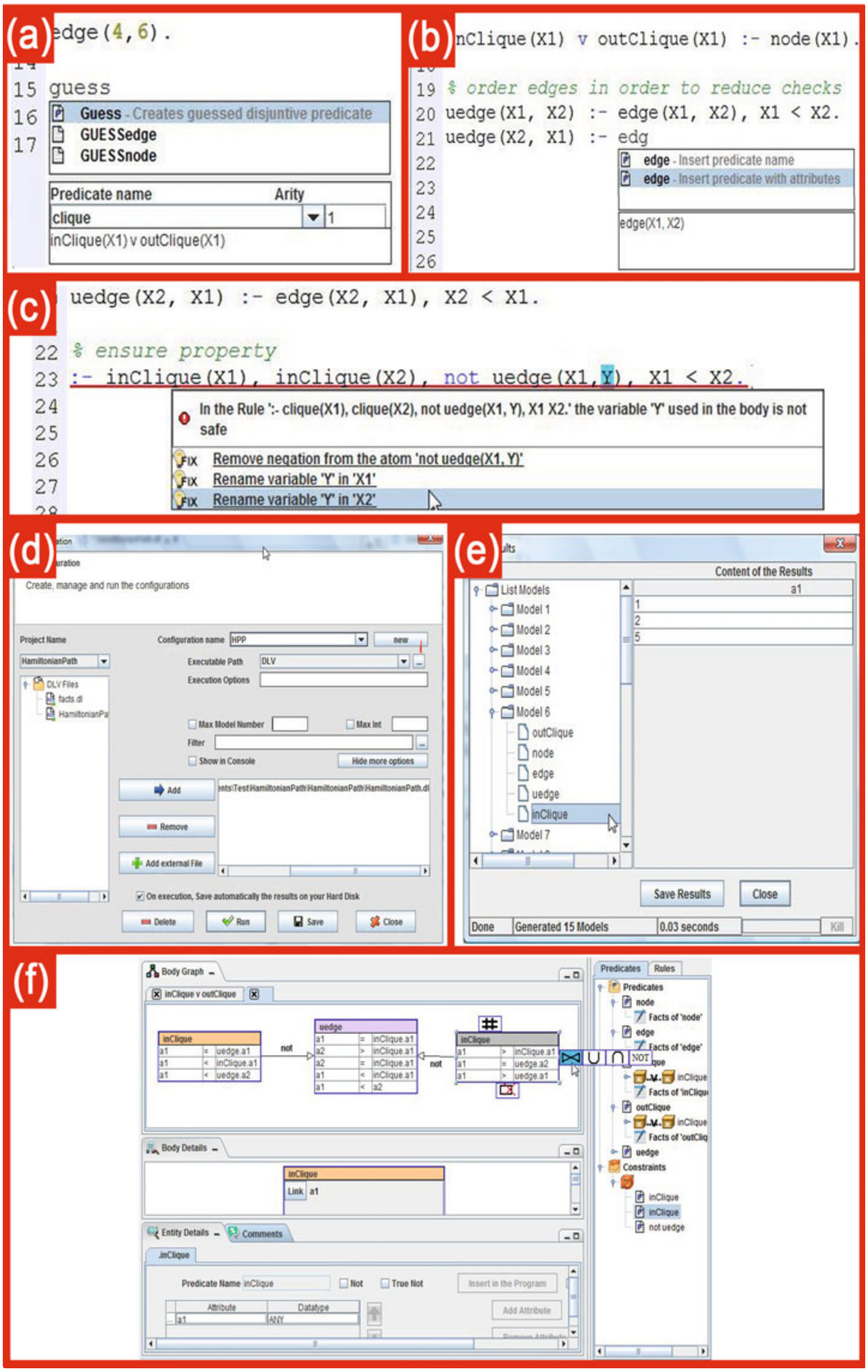
4 Usage Example

In the following we report an usage example by describing how to exploit *ASPIDE* for encoding the maximum clique problem. The maximum clique is a classical *NP-complete* problem in graph theory requiring to find a clique (a complete subgraph) with the largest number of vertices in an undirected graph. Suppose that the graph G is specified by using facts over predicates *node* (unary) and *edge* (binary), then the following program solves the problem:

```
% Guess the clique
r1: inClique(X1) v outClique(X1) :- node(X1).
% Order edges in order to reduce checks
r2: uedge(X1,X2) :- edge(X1,X2), X1 < X2.
r3: uedge(X2,X1) :- edge(X1,X2), X2 < X1.
% Ensure property.
r4: :- inClique(X1), outClique(X2), not uedge(X1,X2), X1 < X2.
r5: :~ outClique(X2).
```

The disjunctive rule ($\mathtt{r}_1$) guesses a subset S of the nodes to be in the clique, while the rest of the program checks whether S constitutes a clique, and the weak constraint maximizes the size of S. Here, an auxiliary predicate *uedge* exploits an ordering for reducing the time spent in checking.

Suppose that we have already created a new file named *clique.dl*, which is open in the text editing area. To write the disjunctive rule $\mathtt{r}_1$, we exploit the code template *guess*.



```
edge(4,6).

guess
```

Guess - Creates guessed disjuntive predicate
GUESSedge
GUESSnode

Predicate name: clique
Arity: 1
inClique(X1) v outClique(X1)

```
nClique(X1) v outClique(X1) :- node(X1).

% order edges in order to reduce checks
uedge(X1, X2) :- edge(X1, X2), X1 < X2.
uedge(X2, X1) :- edg
```

edge - Insert predicate name
edge - Insert predicate with attributes
edge(X1, X2)

```
uedge(X2, X1) :- edge(X2, X1), X2 < X1.

% ensure property
:- inClique(X1), inClique(X2), not uedge(X1,Y), X1 < X2.
```

In the Rule ':- clique(X1), clique(X2), not uedge(X1, Y), X1 X2.' the variable 'Y' used in the body is not safe

FIX Remove negation from the atom 'not uedge(X1, Y)'
FIX Rename variable 'Y' in 'X1'
FIX Rename variable 'Y' in 'X2'

Create, manage and run the configurations

Project Name: HamiltonianPath
Configuration name: HPP
Executable Path: DLV
Execution Options:
Max Model Number
Max Int
Filter
Show in Console
DLV Files: facts.dl, HamiltonianPa
ents\Test\HamiltonianPath\HamiltonianPath\HamiltonianPath.dl
On execution, Save automatically the results on your Hard Disk

Content of the Results

a1
1
2
5

List Models: Model 1, Model 2, Model 3, Model 4, Model 5, Model 6 (outClique, node, edge, uedge, inClique), Model 7

Done | Generated 15 Models | 0.03 seconds

inClique v outClique

inClique		
a1	=	uedge.a1
a1	<	inClique.a1
a1	<	uedge.a2

uedge		
a1	=	inClique.a1
a2	>	inClique.a1
a2	=	inClique.a1
a1	<	inClique.a1
a1	<	a2

inClique		
a1	>	inClique.a1
a1	=	uedge.a2
a1	>	uedge.a1

inClique	
Link	a1

Predicate Name: inClique

Attribute	Datatype
a1	ANY

Fig. 2. Using *ASPIDE*

We type the key word *guess* and press CTRL-Space: the text editor opens a popup window where we indicate *clique*; *ASPIDE* auto-composes the head of the disjunctive rule (a preview is displayed on the bottom of the popup, see Fig. 2(a)). After that, we write the predicate *node* in the body. Rules $\mathtt{r}_2$, $\mathtt{r}_3$, $\mathtt{r}_4$ and $\mathtt{r}_5$ are easily written by exploiting the auto-completion (see Fig. 2(b)): atoms are composed by typing only the initial part of the predicate names, and variables are automatically added.

Now, Suppose that a safety error is revealed in rule $\mathtt{r}_4$ (see Fig. 2(c)) since we write variable *Y* instead of *X2* in the negated literal *uedge*; then, the text editor promptly signals the error with a rule highlighting. By double-clicking on the wrong rule, the system will open a popup window reporting the error message and a list of possible quick fixes. With a simple click on the third proposal, variable *Y* is renamed in *X2*, and the problem is easily detected and fixed. Now, we can execute the program by setting a run configuration. To open the run configuration window (Fig. 2(d)), we select *Show Run Configuration* from the menu *Execute*. Using that window we choose the program file *clique.dl* and select DLV as solver. Finally we click on the *Run* button and a user-friendly window shows the results (see Fig. 2(e)).

Note that, *ASPIDE* supports many different ways of creating and modifying logic programs and files. For instance, the same encoding can be graphically composed by exploiting the visual editor (see Fig. 2(f)). For respecting the space constraints, we reported only one of the possible combinations of commands and shortcuts that can be exploited for writing the encoding of the considered problem. The reader can try *ASPIDE* by downloading it from the system website `http://www.mat.unical.it/ricca/aspide`.

5 Related Work

Different tools for developing ASP programs have been proposed up to now, including editors and debuggers [32,33,34]. Moreover, people from the logic programming community, and especially Prolog programmers are already exploiting tools for assisted program development. Some support to the development of logic programs is also present in environments conceived for logic-based ontology languages, which, besides graphical ontology development, also allow for writing logic programs (e.g., to reason on top of the knowledge base). To the best of our knowledge the systems which are closer to our are the following:

– *OntoDLV* [44], an ASP-based system for ontology management and reasoning on top of ontologies.
– *OntoStudio* (`http://www.ontoprise.de`), a commercial modeling environment for the creation and maintenance of ontologies; it also allows for writing logic programs according with to F-Logic molecules syntax.
– *DES* (`http://www.fdi.ucm.es/profesor/fernan/des/`): a deductive database system that supports both Datalog and SQL; it supports querying, debugging and testing of Datalog programs, and supports several DBMSs.
– *DLV!sual* (`http://thp.io/2009/dlvisual`): a GUI frontend for DLV, which allows for developing programs as well as graphically-browsing the answer sets;

- *Visual DLV* [30]: a graphical integrated environment for developing DLV programs; it helps the programmers during the development phases, supports the interaction with external DBMS and features a naïve debugging tool;
- *Digg* (`http://www.ezul.net/2010/09/gui-for-dlv.html`): a simple Java application conceived for learning ASP and experimenting with DLV;
- *APE* [31]: an Eclipse plug-in that allows users for writing ASP programs in the lparse/gringo language;
- *J-Prolog*(`http://www.trix.homepage.t-online.de/JPrologEditor`): an IDE for the Prolog language, written in Java.
- *ProDT* (`http://prodevtools.sourceforge.net`): an Eclipse plugin for developing Prolog projects.
- *PDT* (`http://roots.iai.uni-bonn.de/research/pdt`): an Eclipse plugin for developing Prolog projects.
- *ProClipse* [45]): an Eclipse plugin for developing Prolog projects;
- *Amzi! Prolog* (`http://www.amzi.com/products/prolog_products.htm`: an Eclipse plugin for developing Prolog projects.

People interested in programming, in general, wish to have, in those systems, a set of editing features (which are already available for a long time in development tools for imperative languages) like syntax coloring, syntax highlighting, code completion, error management, quick fix, etc. Debugging, profiling and testing tools, as well as the capability of organizing programs in projects are fundamental for assisting the development of complex applications. Moreover, the declarative nature of logic programming languages makes them good candidates for developing tools which allow for writing programs in a fully graphical way. In Table 1 we compare *ASPIDE* and the above listed tools by separately considering general desirable features. A check symbol indicates that a system provides (in a more or less sophisticated way) a feature. We first note that *ASPIDE* is the most complete proposal, followed by the commercial product *OntoStudio*; and, if we restrict our attention to competing systems tailored for ASP, APE follows *ASPIDE*.

Note that, execution results are reported by most systems only in a textual form (only *ASPIDE*, OntoDLV and OntoStudio offer a graphical view of them in intuitive tables), the outline of the program is often missing, and the execution of systems/solvers is not handled in an effective way. Moreover, only *ASPIDE*, OntoStudio and APE show the

dependency graphs in a graphical way, and the detection of errors during the editing phase, as well as (some form of) debugging and testing are offered by a few systems. Interaction with databases, often required by applications, is supported by only 5 systems out of 13 (data integration only by two).

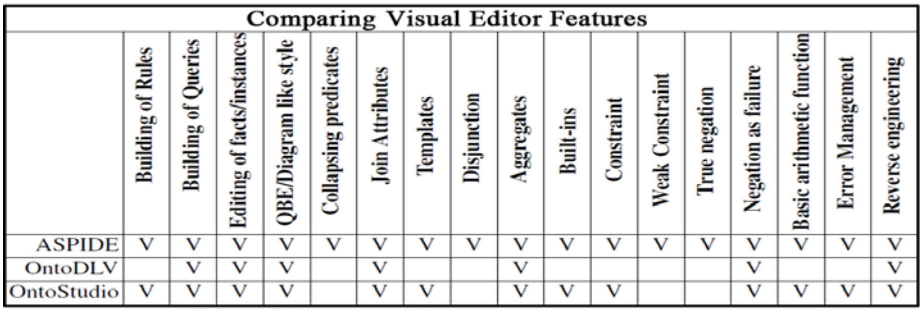
Conversely, text editing is supported by all systems, and in order to provide a more precise picture regarding this central feature we have deepened the analysis by considering, also, more advanced editing functionalities and support for project management (see Table 1). Also in this case, *ASPIDE* is the system offering more features. Surprisingly, *OntoStudio* lacks advanced text editing features, which are conversely provided by the more mature environments for Prolog. It is worth noting that, systems based on the Eclipse platform, which eases the development of text editors, provide quite a number of editing and project management features (see, e.g., APE). In general, many

Table 1. Logic programming tools comparison

General Features	ASPIDE	OntoDLV	OntoStudio	DES	DLV!SUAL	VISUAL DLV	Digg	APE	J-Prolog	ProDT	PDT	ProClipse	Amzi! Prolog
Comparing General Features													
Text Editor	V	V	V	V	V	V	V	V	V	V	V	V	V
Visual Editor	V	V	V										
Project Management	V		V	V				V		V	V	V	V
File Content Outline	V	V	V			V		V		V	V	V	V
Test Suite	V		V	V						V			
Integration with databases	V		V	V		V						V	
Debugger	V		V	V		V			V	V	V		V
Command line				V					V	V	V	V	V
Visual Dependency Graph	V		V					V					
Profiler/Tracer	V			V					V	V	V		
Global Error Console	V	V	V			V		V		V	V	V	V
Textual Result Visualization	V	V		V	V	V	V	V	V	V	V	V	V
User Friendly Result Visualization	V	V	V										
Visual Workflow Definition	V												
Detect Error on editing	V		V			V				V	V	V	V
Dynamic Layout	V		V					V		V	V	V	V
Refactor variables and predicates	V		V										
Configuration of the execution	V		V	V	V	V		V		V	V		V
Data Integration	V		V										
Datatype Management	V	V	V	V		V							
Comparing Text Editor Features													
Text Coloring	V			V	V			V	V	V	V	V	V
Parenthesis pair highlighter	V				V			V		V	V	V	
Token pair highlighter	V				V		V	V		V	V	V	
Undo/Redo	V				V			V	V	V	V	V	V
Syntactic Error/Warning highlighter	V							V		V	V	V	
Semantic Error/Warning highlighter	V									V	V	V	
Find/Replace	V			V	V					V	V	V	V
Quick Fix	V									V	V	V	
Auto completion	V					V		V		V	V	V	V
Code Templates	V					V				V	V		
Dynamic Code Template Definition	V										V		
Code Annotation	V												
Automatic Code Indentation	V												
Text Hover (quick info of predicates)												V	
Code Documentation (like JavaDoc)	V											V	
Comparing Project Management Features													
Multi Project	V		V					V		V	V	V	V
Multi File	V		V	V	V	V	V	V	V	V	V	V	V
Sub Folder Organization	V		V					V		V	V	V	V

existing tools provide syntax coloring, but few systems for ASP support code completion and quick fix of errors. It is quite strange that very basic features like undo/redo and find/replace are not supported by all systems. A particular mention merits the Source-to-Source transformation feature, which allows for translating one language to another. Actually, only *OntoStudio* and *DES* support this feature to translate, respectively, from an ontology language to an other one (e.g., F-Logic and RDF) and from Datalog to SQL. Although every considered system supports a textual editor, only *ASPIDE*, OntoDLV and OntoStudio offer a complete graphical editing environment for writing programs. Thus, in Table 2 we report only the systems allowing for graphic composition

Table 2. Visual editing tools: features and language constructs



Comparing Visual Editor Features

	Building of Rules	Building of Queries	Editing of facts/instances	QBE/Diagram like style	Collapsing predicates	Join Attributes	Templates	Disjunction	Aggregates	Built-ins	Constraint	Weak Constraint	True negation	Negation as failure	Basic arithmetic function	Error Management	Reverse engineering
ASPIDE	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V
OntoDLV		V	V	V		V			V					V			V
OntoStudio	V	V	V	V		V	V		V	V	V			V	V	V	V

of programs. In this case we consider also supported language constructs. Focusing on ASP-based systems *ASPIDE* easily beats *OntoDLV*, which supports only queries. *OntoStudio*, which supports a different logic language, clearly misses many ASP-specific constructs, but provides a rich environment that supports ontology constructs.

6 Conclusion

This paper presents *ASPIDE*, a comprehensive IDE for ASP, combining several tools for supporting the entire life-cycle of logic programs development. A key feature of *ASPIDE* is its rich set of assisted program composition features, like dynamic syntax highlighting, on-line syntax correction, autocompletion, code-templates, quick-fixes, refactoring, visual editing, program browsing etc. Moreover, *ASPIDE* integrates several graphic tools *tailored for ASP*, including: fully-graphical program composition, debugging, profiling, project management, DBMS access, execution configuration, and user-friendly output-handling. Comparing several softwares for developing logic programs, *ASPIDE* is one of the most complete solutions available in the present state of the art.

As far as future work is concerned, we plan to extend *ASPIDE* by improving / introducing additional dynamic editing instruments, and graphic tools. In particular, we plan to enrich the interface with source-to-source transformation (e.g., for supporting seamless conversions from multiple ASP dialects); and, to develop/extend the input/output interfaces for handling multiple ASP solvers. Moreover, we are improving the testing tool by including more advanced approaches such as [43]. We are currently using *ASPIDE* in a logic programming course of the University of Calabria to assess the applicability of the system for teaching ASP.

Acknowledgments. This work has been partially supported by the Calabrian Region under PIA (Pacchetti Integrati di Agevolazione industria, artigianato e servizi) project DLVSYSTEM approved in BURC n. 20 parte III del 15/05/2009 - DR n. 7373 del 06/05/2009.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC 9, 365–385 (1991)
2. Lifschitz, V.: Answer Set Planning. In: ICLP 1999, pp. 23–37 (1999)
3. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS 22(3), 364–418 (1997)
4. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL 7(3), 499–562 (2006)
5. Simons, P.: Smodels Homepage (since 1996), `http://www.tcs.hut.fi/Software/smodels/`
6. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI 138, 181–234 (2002)
7. Zhao, Y.: ASSAT homepage (since 2002), `http://assat.cs.ust.hk/`
8. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI 2002, Edmonton, Alberta, Canada. AAAI Press / MIT Press (2002)
9. Babovich, Y., Maratea, M.: Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs (2003), `http://www.cs.utexas.edu/users/tag/cmodels.html`
10. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007, pp. 386–392 (2007)
11. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. ACM TOCL 7(1), 1–37 (2006)
12. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 447–451. Springer, Heidelberg (2005)
13. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008), Sydney, Australia, pp. 422–432. AAAI Press, Menlo Park (2008)
14. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 3–17. Springer, Heidelberg (2007)
15. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The Second Answer Set Programming Competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
16. Balduccini, M., Gelfond, M., Watson, R., Nogueira, M.: The USA-Advisor: A Case Study in Answer Set Planning. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 439–442. Springer, Heidelberg (2001)
17. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Logic-Based Artificial Intelligence, pp. 257–279. Kluwer, Dordrecht (2000)
18. Baral, C., Uyan, C.: Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 186–199. Springer, Heidelberg (2001)
19. Friedrich, G., Ivanchenko, V.: Diagnosis from first principles for workflow executions. Tech. Rep., `http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf`
20. Franconi, E., Palma, A.L., Leone, N., Perri, S.: Census Data Repair: A Challenging Application of Disjunctive Logic Programming. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 561–578. Springer, Heidelberg (2001)
21. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog Decision Support System for the Space Shuttle. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 169–183. Springer, Heidelberg (2001)

22. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: SIGMOD 2005, Baltimore, Maryland, USA, pp. 915–917. ACM Press, New York (2005)
23. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. In: CUP (2003)
24. Bardadym, V.A.: Computer-Aided School and University Timetabling: The New Wave. In: Burke, E.K., Ross, P. (eds.) PATAT 1995. LNCS, vol. 1153, pp. 22–45. Springer, Heidelberg (1996)
25. Grasso, G., Iiritano, S., Leone, N., Ricca, F.: Some DLV Applications for Knowledge Management. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 591–597. Springer, Heidelberg (2009)
26. Grasso, G., Leone, N., Manna, M., Ricca, F.: Gelfond Festschrift. LNCS, vol. 6565. Springer, Heidelberg (2010)
27. Dovier, A., Erdem, E.: Report on application session @lpnmr09 (2009), http://www.cs.nmsu.edu/ALP/2010/03/report-on-application-session-lpnmr09/
28. De Vos, M., Schaub, T. (eds.): SEA 2007: Software Engineering for Answer Set Programming, vol. 281. CEUR (2007), http://CEUR-WS.org/Vol-281/
29. De Vos, M., Schaub, T. (eds.): SEA 2009: Software Engineering for Answer Set Programming, vol. 546. CEUR (2009), http://CEUR-WS.org/Vol-546/
30. Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA 2007), pp. 86–100 (2007)
31. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: An AnsProlog* Environment. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA 2007), pp. 101–115 (2007)
32. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: That is Illogical Captain! The Debugging Support Tool spock for Answer-Set Programs: System Description. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA 2007), pp. 71–85 (2007)
33. Brain, M., De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In: Proceedings ASP 2005 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
34. El-Khatib, O., Pontelli, E., Son, T.C.: Justification and debugging of answer set programs in ASP. In: Proceedings of the Sixth International Workshop on Automated Debugging. ACM, New York (2005)
35. Oetsch, J., Pührer, J., Tompits, H.: Catching the ouroboros: On debugging non-ground answer-set programs. In: Proc. of the ICLP 2010 (2010)
36. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging asp programs by means of asp. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
37. Calimeri, F., Ianni, G., Ricca, F.: The third answer set programming system competition (since 2011), https://www.mat.unical.it/aspcomp2011/
38. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP 8, 129–165 (2008)
39. Febbraro, O., Reale, K., Ricca, F.: A Visual Interface for Drawing ASP Programs. In: Proc. of CILC 2010, Rende, CS, Italy (2010)
40. Calimeri, F., Ianni, G.: Template programs for Disjunctive Logic Programming: An operational semantics. AI Communications 19(3), 193–206 (2006)
41. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A Visual Tracer for DLV. In: Proc. of SEA 2009, Potsdam, Germany (2009)

42. Ricca, F.: The DLV Java Wrapper. In: ASP 2003, Messina, Italy, pp. 305–316 (2003), `http://CEUR-WS.org/Vol-78/`
43. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence, pp. 951–956. IOS Press, Amsterdam (2010)
44. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: an ASP-based system for enterprise ontologies. Journal of Logic and Computation (2009)
45. Bendisposto, J., Endrijautzki, I., Leuschel, M., Schneider, D.: A Semantics-Aware Editing Environment for Prolog in Eclipse. In: Proc. of WLPE 2008 (2008)

ASP-Prolog for Negotiation among Dishonest Agents

Ngoc-Hieu Nguyen[1], Tran Cao Son[1], Enrico Pontelli[1], and Chiaki Sakama[2]

[1] New Mexico State University
{nhieu,tson,epontell}@cs.nmsu.edu
[2] Wakayama University
sakama@sys.wakayama-u.ac.jp

Abstract. This paper describes a platform to develop negotiating agents, whose knowledge and rules of behavior are represented as Abductive Logic Programs. The platform implements a flexible negotiation framework. Negotiating agents can operate with multiple goals and incomplete knowledge, and dynamically modify their goals depending on the progress of the negotiation exchanges. Differently from other frameworks, agents can operate dishonestly, by generating false statements or statements that are not substantiated by the agents' knowledge. The proposed platform has been implemented using the ASP-Prolog platform.

1 Introduction

Real-world interactions among agents often require the ability to perform *negotiation*—i.e., perform sequences of offers/counter-offers to reach a consensus about possible exchanges of knowledge and/or goods. The issue of modeling negotiation in multi-agent systems is an important area of research that has received significant attention, e.g., [1,5,6,9,11,14]. Nevertheless, of the many frameworks and theories proposed to model negotiation among agents, relatively few are capable of handling *incomplete information* (e.g., [3,13]) and even fewer have considered the case of agents that are lying or misleading other agents during the negotiation process [15].

We have recently [12] developed a theoretical framework that enables modeling negotiation among two agents, where each agent can introduce deception in the generation of offers/counter-offers, and he has incomplete knowledge about the other agent and preferences on the choice of possible counter-offers. The proposal builds on the use of *Abductive Logic Programming (ALP) with preferences* [10] to encode the knowledge bases of the agents and their strategies for negotiation (including strategies to introduce deception). It enables to model interactions such as the one in the following example.

Example 1. A buyer agent and a seller agent negotiate the purchase/sale of a camera. The buyer starts the negotiation by indicating the desire to purchase a camera produced by maker C, at the lowest price and of good quality. The seller responds that the model A is produced by C and has good quality, and it is sold at discounted price to students—in this case, the seller may mislead the buyer, by stating knowledge about the quality of product A that he/she may not have. The buyer could honestly respond that he is not a student. The seller may attempt to draw the attention on product B, a camera produced by maker D, of good quality, sold at discounted priced if paid in cash—in this case, the seller is lying, as he/she knows that B is a poor quality product. The buyer is aware of the poor quality, and refuses to accept the offer to purchase B at the discounted price. One more time, the seller focuses on B by offering a further discount if the buyer joins the seller's mailing list. At that point, the buyer accepts the offer (but he/she really has no intention to subscribe to the mailing list, thus he/she lies about it). $\square$

In this paper, we describe a platform that implements a variant of ALP necessary to support the negotiation process, and its use in modeling agents that can negotiate in presence of incomplete knowledge, deception, and preferences. The framework is realized using a combination of answer set programming and Prolog, as provided by the ASP-Prolog system [8].

2 Background

2.1 Abductive Logic Programming (ALP), Preferences, and Disinformation

We use an extension of abductive logic programs defined in [10]. An *abductive program* is a pair (P^r, P^a) where P^r and P^a are disjunctive logic programs. The rules in P^a are referred to as *abducibles*. A set of literals S is a *belief set* of (P^r, P^a) if S is an answer set of $P^r \cup E$ for some $E \subseteq P^a$. (P^r, P^a) is *consistent* if it has a belief set; otherwise, it is inconsistent. For a program (P^r, P^a) and a set of rules X, $P \cup^r X$ (resp. $P \cup^a X$) denotes the abductive program $(P^r \cup X, P^a)$ (resp. $(P^r, P^a \cup X)$). Given a set of literals S, we denote $S^\neg = \{\neg\ell \mid \ell \in S\}$ ($\neg\neg a$ represents the atom a).

For the discussion in the following sections, we associate a name n_r to each rule r and freely use the name to represent the rule[1]. When multiple sets of abducible rules can be used to generate belief sets of a program, a *preference* relation among abducibles, in the form of $\mathit{prefer}(n_1, n_2)$, can be introduced, allowing us to define a preference relation among belief sets. It is assumed that prefer is a transitive and asymmetric relation among abducibles of a program. The relation prefer is also extended to define a preference relation among sets of abducible rules as follows: for $Q_1, Q_2 \subseteq P^a$, Q_1 is preferred to Q_2 if either (*i*) $Q_1 \subseteq Q_2$ or (*ii*) there exist $n_1 \in Q_1 \setminus Q_2$ and $n_2 \in Q_2 \setminus Q_1$ such that $\mathit{prefer}(n_1, n_2)$ holds and there exists no $n_3 \in Q_2 \setminus Q_1$ such that $\mathit{prefer}(n_3, n_1)$ holds. In turn, this allows the comparison of belief sets of a program $P = (P^r, P^a)$; if S_1 (resp. S_2) is a belief set of P obtained from $P^r \cup Q_1$ (resp. $P^r \cup Q_2$), then S_1 is *preferred* to S_2 if Q_1 is preferred to Q_2. A belief set S of P is a *most preferred* belief set if there is no belief set S' of P that is preferred to S.

Example 2. Let us consider a seller agent s whose sale knowledge is encoded by an abductive program with preferences $P_s = (P^r_s, P^a_s)$ as follows:

[1] We omit the rule names when not needed in the discussion.

- P_s^r consists of the rules:

$$\begin{array}{rlrl}
senior\_customer & \leftarrow age \geq 65 & product_A & \leftarrow \\
student\_customer & \leftarrow student & product_B & \leftarrow \\
\neg quality_B & \leftarrow product_B & prefer(n_1, n_i) & \leftarrow \text{(for } i \in \{2,3,4,5\}) \\
maker_C & \leftarrow product_A & prefer(n_i, n_5) & \leftarrow \text{(for } i \in \{2,3,4\}) \\
maker_D & \leftarrow product_B & & \leftarrow high\_pr, low\_pr \\
bargain & \leftarrow product_B & & \leftarrow high\_pr, lowest\_pr \\
sale & \leftarrow product_A, \mathbf{price_1} & & \leftarrow low\_pr, lowest\_pr \\
sale & \leftarrow product_B, \mathbf{price_2} & & \leftarrow \ not\ sale
\end{array}$$

where $\mathbf{price_1} \in \{\, high\_pr,\ low\_pr \,\}$ and $\mathbf{price_2} \in \{\, low\_pr, lowest\_pr,\ high\_pr\}$. P_s^r defines various types of customers and some features of the products. It also states the sales conditions and the preferences among the abducible rules of the seller. The constraints on the prices indicate that the seller sells a product for only one price.
- $P_s^a = H_s \cup R_s$ where $H_s = \{\, age \geq 65,\ student,\ cash,\ mail\_list \,\}$ and R_s consists of the following rules

$$\begin{array}{llll}
n_1 : high\_pr \leftarrow & & & \\
n_2 : low\_pr & \leftarrow senior\_customer & n_4 : low\_pr & \leftarrow bargain,\ cash \\
n_3 : low\_pr & \leftarrow student\_customer & n_5 : lowest\_pr & \leftarrow mail\_list,\ cash
\end{array}$$

Intuitively, each belief set of s represents one possible way to sell a product. It is easy to see that any belief set of P_s must contain at least one of the literal $high\_pr$, low_pr, or $lowest\_pr$; the most preferred belief set of P_s contains $high\_pr$. □

Dishonest agents are those who use intentionally false or inaccurate information. For an abductive program $P = (P^r, P^a)$, a pair of disjoint sets of literals (L, B) is called *disinformation* w.r.t. P if

- L represents *lies* [7], literals that are known to be false—i.e., $\forall l \in L$, $\neg l$ belongs to every belief set of P.
- B represents *bullshit (BS)* [4], literals that are unknown—i.e., $\forall l \in B$, neither l nor $\neg l$ belongs to any belief set of P.

Given a disinformation (L, B) w.r.t. a program $P = (P^r, P^a)$, we define

$$\begin{array}{rl}
I & = \{\, r \mid r \in P^r \text{ and } head(r) \cap L^{\neg} \neq \emptyset\}, \\
\Phi & = \{\, prefer(n_i, n_j) \mid n_i \in I \text{ and } n_j \in P^a\} \ \cup \\
 & \quad \{\, prefer(n_b, n_l) \mid n_b \in B \text{ and } n_l \in L\} \ \cup \\
 & \quad \{\, prefer(n_j, n_t) \mid n_j \in P^a \cup I \text{ and } n_t \in (L \cup B)\}.
\end{array}$$

Φ represents that (i) any rule from P^r is preferred to hypotheses in P^a, (ii) bullshit is preferred to lies, and (iii) honest information is preferred to dishonest one. The abductive program $\delta(P, L, B) = (P^r \setminus I \cup \Phi, P^a \cup I \cup L \cup B)$ is called the *Abductive Logic program with Disinformation (ALD-program)* (L, B) w.r.t. P.

Example 3. Assume that s from Example 2 can claim that both products A and B are of good quality, if needed to make a sale—i.e., s would lie about $quality_B$ and BS about $quality_A$. This is described by the disinformation: (L_s, B_s)=$(\{quality_B\}, \{quality_A\})$.

The ALD-program (L_s, B_s) w.r.t. P_s is $\delta(P_s, L_s, B_s) = (\delta P_s^r, \delta P_s^a)$ where δP_s^a is equal to $P_s^a \cup L_s \cup B_s$ plus the rule n_6 defined as $\neg quality_B \leftarrow product_B$, and $\delta P_s^r = P_s^r \setminus \{\neg quality_B \leftarrow product_B\}$ plus the preferences

- $prefer(n_6, n_i)$ for $i = 1, \ldots, 5$ and $prefer(n_6, n_h)$ where $n_h \in H_s$ (each fact in H_s is considered as a rule with the same name);
- $prefer(n_h, n_t)$ and $prefer(n_j, n_t)$ for $j = 1, \ldots, 6$, $n_h \in H_s$ and $n_t \in (L_s \cup B_s)$;
- $prefer(n_b, n_l)$ for $n_b \in B$ and $n_l \in L$. □

Because $\delta(P, L, B)$ is an abductive program, belief sets of $\delta(P, L, B)$ and the preferences among them are defined as before. For a set of belief sets Σ of $\delta(P, L, B)$, we say that $M \in \Sigma$ is a most preferred belief set of Σ if there exists no $M' \in \Sigma$ such that M' is preferred to M.

2.2 Negotiation Knowledge Base (n-KB)

A *negotiation knowledge base (n-KB)* is a tuple $\langle P, L, B, H, N^{\prec}\rangle$ where P is an abductive program encoding the agent's beliefs and rules of conduct; (L, B) is a disinformation with respect to P, representing possibly false information that the agent may use in order to obtain a deal; H is a set of assumptions about the other party; $N^{\prec}$ is a set of literals whose members represent goals (issues) that he would like to negotiate, along with a strict partial order $\prec$ among them.

Example 4. Continuing with the previous example, assume that the n-KB of the agent s be $K_s = \langle P_s, L_s, B_s, H_s, N_s^{\prec}\rangle$ where P_s, H_s, and (L_s, B_s) are given in Examples 2 and 3, respectively, and $N_s^{\prec}$ is the set of possible prices that the seller is willing to negotiate about: $N_s^{\prec} = \{\, high\_pr,\ low\_pr,\ lowest\_pr \,\}$ with $lowest\_pr \prec low\_pr \prec high\_pr$. This indicates that the seller prefers $high\_pr$ over low_pr and $lowest\_pr$. □

Example 5. An n-KB $K_b = \langle P_b, L_b, B_b, H_b, N_b^{\prec}\rangle$ with $P_b = (P_b^r, P_b^a)$ for the buyer b:

- P_b^r consists of

$$\begin{array}{rlr} purchase \leftarrow & product_X,\ quality_X,\ \mathbf{price_3} & prefer(n_1, n_2) \leftarrow \\ \leftarrow & not\ purchase & prefer(n_2, n_3) \leftarrow \\ \neg student \leftarrow & & prefer(n_1, n_3) \leftarrow \\ \leftarrow & low\_pr, lowest\_pr & cash \leftarrow \end{array}$$

  where $X \in \{A, B\}$ and $\mathbf{price_3} \in \{\, low\_pr,\ lowest\_pr \,\}$.
- $H_b = \{\, quality_A,\ quality_B,\ maker_C,\ maker_D,\ product_A, product_B\}$.
- $N_b^{\prec}$=$\{\, low\_pr,\ lowest\_pr \,\}$ with $\prec$= $\{\, low\_pr \prec lowest\_pr \,\}$.
- $P_b^a = H_b \cup R_b$ where R_b consists of

$$n_1 : lowest\_pr \leftarrow maker_C \quad n_2 : lowest\_pr \leftarrow maker_D \quad n_3 : low\_pr \leftarrow maker_C$$

The set H_b represents properties of products that the buyer needs to check and $N_b^{\prec}$ specifies that the buyer prefers to pay the lowest price. Suppose that the buyer does not want to be on the seller's mailing list but could pretend to join it if it works to his/her advantage. S/he decides to use the disinformation $(L_b, B_b) = (\emptyset, \{mail\_list\})$ w.r.t. P_b. Thus, he/she will use the ALD-program $\delta(P_b, L_b, B_b)$ in his/her negotiation. □

2.3 Negotiations Using n-KBs

In this section, we review the definitions of negotiation among dishonest agents using n-KBs [12]. We consider negotiations involving two agents a and b, whose n-KBs are K_a and K_b. We assume that K_a and K_b share the same language and the set of assumptions in K_a is disjoint from the set of assumptions in K_b. A negotiation contains several rounds, where the two agents alternate in generating proposals to the other agent.

Intuitively, for an agent a, a proposal is a tuple $\langle G, S, R\rangle$ stating that the goal of a is to negotiate to achieve G. The proposal is supported by a belief set M. By making the proposal, a indicates assumptions that she has made about the receiver of the proposal (the set S), as well as information about her/himself (R) that the receiver of the proposal will have to respect. Formally, a *proposal* for $G \subseteq N$ w.r.t. $K = \langle P, L, B, H, N^{\prec}\rangle$ is a tuple $\gamma = \langle G, S, R\rangle$ where $\delta(P, L, B) \cup^r \mathit{Goal}(G)$[2] has a belief set M such that $S = M \cap H$, and $R \subseteq M \setminus H$. We refer to G, S, R, and M as the *goal*, *assumptions*, *conditions*, and *support* of $\langle G, S, R\rangle$, respectively. The proposal is *honest* if $M \cap (L \cup B) = \emptyset$; it is *deceptive* if $M \cap L \neq \emptyset$; and it is *unreliable* if $M \cap B \neq \emptyset$.

Let $\gamma = \langle G, S, R\rangle$ be a proposal from an agent b to an agent a, the latter described by the n-KB $K_a = \langle P, L, B, H, N^{\prec}\rangle$. Let $\delta Q = \delta(P, L, B) \cup^r \mathit{Goal}(G)$.

- γ is *acceptable* w.r.t. K if δQ has a belief set M such that $S \subseteq M$ and $M \cap H \subseteq R \cap H$. We say that γ is acceptable *without disinformation* if $M \cap (L \cup B) = \emptyset$, *with disinformation*, otherwise.
- γ is *rejectable* if δQ is inconsistent.
- γ is *negotiable*, otherwise.

Example 6. For K_s and K_b from Examples 4 and 5:
$\langle\{high\_pr\}, \{product_A\}, \emptyset\rangle$ is acceptable without disinformation w.r.t. K_s, as $\delta(P_s \cup^r \mathit{Goal}(\{high\_pr\}), L_s, B_s)$ has a disinformation-free belief set $\{high\_pr, product_A\} \subseteq M$.
$\langle\{high\_pr\}, \{product_A, quality_A\}, \emptyset\rangle$ is acceptable with disinformation w.r.t. K_s as $\delta(P_s \cup^r \mathit{Goal}(\{high\_pr\}), L_s, B_s)$ has a belief set M containing $high\_pr$, $product_A$, and each belief set with this property contains $quality_A$.
$\langle\{low\_pr\}, \{product_B, maker_D, quality_B\}, \emptyset\rangle$ is a negotiable proposal w.r.t. K_s since $\delta(P_s \cup^r \mathit{Goal}(\{low\_pr\}), L_s, B_s)$ has a belief set containing its assumptions but requires at least one of the sets $\{student\}$, $\{age \geq 65\}$, or $\{cash\}$.
$\langle\{high\_pr\}, \emptyset, \{product_A, maker_C, quality_A\}\rangle$ is a rejectable proposal w.r.t. K_b because $\delta(P_b \cup^r \mathit{Goal}(\{high\_pr\}), L_b, B_b)$ has no belief set containing $high\_pr$. □

The receiver of a proposal generates a response. Let K_a be the n-KB of agent a and $\gamma_b = \langle G, S, R\rangle$ be a proposal by b w.r.t. its n-KB K_b. A *response* to γ_b by a is: *(i)* A proposal

[2] $\mathit{Goal}(G) = \{\leftarrow \mathit{not}\ \ell \mid \ell \in G\}$.

$\gamma_a = \langle G', S', R' \rangle$ w.r.t. K_a; **or** *(ii)* $\langle \top, \emptyset, \emptyset \rangle$, denoting *acceptance of the proposal* if γ_b is acceptable w.r.t. K_a; **or** *(iii)* $\langle \bot, \emptyset, \emptyset \rangle$, denoting *rejection of the proposal.* Intuitively, a negotiation is a series of responses between two agents, who, in alternation, take into consideration the other agent's response and put forward a new response; this can be either accept, reject, or a new proposal that may involve explanations of why the latest proposal (of the other agent) was not acceptable.

Formally, a *negotiation* between a and b, starting with a, is a possible infinite sequence of proposals $\omega_1, \ldots, \omega_i, \ldots$ where $\omega_i = \langle G_i, S_i, F_i \rangle$, ω_{2k+1} is a proposal w.r.t. K_a ($k \geq 0$), ω_{2k} is a proposal w.r.t. K_b ($k \geq 1$), and ω_{i+1} is a response to ω_i for every $i \geq 1$. A negotiation *ends* at i if $\omega_i = \langle \top, \emptyset, \emptyset \rangle$ or $\omega_i = \langle \bot, \emptyset, \emptyset \rangle$. When $G_i \neq G_{i+2}$, we say that a *goal change* has occurred for the agent who proposes ω_i.

Example 7. Consider the seller s and the buyer b agents (Examples 4 and 5).

$b_1 : \langle \{low\_pr\}, \{product_A, quality_A, maker_C\}, \emptyset \rangle$
$s_1 : \langle \{low\_pr\}, \{student\}, \{product_A, quality_A, maker_C\} \rangle$
$b_2 : \langle \{low\_pr\}, \{product_A, quality_A, maker_C\}, \{\neg student\} \rangle$
$s_2 : \langle \{low\_pr\}, \{cash\}, \{product_B, maker_D, quality_B\} \rangle$
$b_3 : \langle \{lowest\_pr\}, \{product_B, maker_D, quality_B\}, \{cash\} \rangle$
$s_3 : \langle \{lowest\_pr\}, \{cash, mail\_list\}, \{product_B, maker_D, quality_B\} \rangle$
$b_4 : \langle \top, \emptyset, \emptyset \rangle$.

The seller bullshits in s_1 and lies in s_2. The buyer lies in b_4. A goal change has occurred at b_3 (for the buyer) and s_3 (for the seller). □

By definition, a negotiation can be infinite. Nevertheless, if agents do not repeat their responses then a negotiation is finite. Given two agents, a negotiation tree encodes all possible negotiations among them. In order to achieve their goals, agents employ *negotiation strategies* to construct their responses, given their n-KBs and a proposal. Detailed discussions on these notions are given in [12]. We present here one strategy. Let $\gamma_b = \langle G, S, R \rangle$ be a proposal by b w.r.t. K_b. A *conscious response* to γ_b by a is

(*i*) A proposal $\gamma_a = \langle G', S', R' \rangle$ w.r.t. K_a with a support M such that $G \preceq G'$, $R \cap H \subseteq S'$, and $S^{\neg} \cap M \subseteq R'$ where M is the support of γ_a, if γ_b is not rejectable w.r.t. K_a; **or**
(*ii*) A proposal $\gamma_a = \langle G', S', R' \rangle$ w.r.t. K_a with a support M such that $G \not\preceq G'$ and $S^{\neg} \cap M \subseteq R'$ where M is a support of γ_a if γ_b is rejectable w.r.t. K_a; **or**
(*iii*) $\langle \top, \emptyset, \emptyset \rangle$, denoting *acceptance of the proposal*, if γ_b is acceptable w.r.t. K_a; **or**
(*iv*) $\langle \bot, \emptyset, \emptyset \rangle$, denoting *rejection of the proposal.*

If the proposal $\langle G, S, R \rangle$ is acceptable, the agent can accept it (case (*iii*)) or negotiate for better options (case (*i*)). If the proposal is negotiable, he can attempt to get a better option (case (*i*)). If the proposal is rejectable, the agent can negotiate for something that is not as good as the current goal (case (*ii*)). In any case, the agent can stop with a rejection (case (*iv*)). An agent generates a new proposal whose goal depends on the goal of the original one, whose assumptions cover the conditions in the original proposal ($R \cap H \subseteq S'$), and whose conditions identify all incorrect assumptions in the original one ($S^{\neg} \cap M \subseteq R'$). An agent who decides to consider preferable proposals, requires that the support for the new proposal is preferred to any support for accepting γ_b.

Example 8. The proposal $\gamma_1 = \langle\{low\_pr\}, \{cash\}, \{product_A, quality_A, maker_C\}\rangle$ ("[s]: I can sell you the $product_A$, made by $maker_C$, and has good quality for low_pr if you pay in cash") is acceptable w.r.t. K_b. The buyer can also respond with $\langle\{lowest\_pr\}, \{product_A, quality_A, maker_C\}, \{cash\}\rangle$ ("[b]: Can I get the $lowest\_pr$?").
The proposal $\gamma_2 = \langle\{low\_pr\}, \{product_A, maker_C\}, \emptyset\rangle$ ("[b]: Can I have $product_A$ from $maker_C$ for low_pr?") is not acceptable but negotiable w.r.t. K_s, since low_pr requires additional assumptions (e.g., $student$ or $age \geq 65$). The seller responds $\langle\{low\_pr\}, \{student\}, \{product_A, maker_C\}\rangle$ ("[s]: Yes, if you are a student.").

The proposal $\gamma_3 = \langle\{high\_pr\}, \emptyset, \{product_A, quality_A, maker_C\}\rangle$ ("[s]: I can sell $product_A$, made by $maker_C$, has good quality, at $high\_pr$") is rejectable w.r.t. K_b, as no rule in K_b derives $high\_pr$. The buyer can reject this proposal or weaken the goal: $\langle\{low\_pr\}, \{product_A, quality_A, maker_C\}, \emptyset\rangle$ ("[b]: Can I get it for low_pr?"). □

We define an agent a with the n-KB $K = \langle P, L, B, H, N^{\prec}\rangle$ to be an *adaptive* agent if it imports the information received during the negotiation into his/her n-KB and keeps this information for the next round of negotiation. Furthermore, an adaptive agent prefers to accept a proposal if a better outcome cannot be achieved. Formally, a is adaptive if for every negotiation $\omega_1, \ldots$, whenever a responds to a proposal $\omega_i = \langle G, S, R\rangle$,

- a responds with $\langle G', S', R'\rangle$, which is a conscious response to $\langle G, S, R\rangle$, and if $\langle G, S, R\rangle$ is acceptable w.r.t. K then G' is preferred to G or $\langle G', S', R'\rangle = \langle\top, \emptyset, \emptyset\rangle$.
- a changes his n-KB to $K' = \langle P \cup^r (R \cap H), L, B, H, N^{\prec}\rangle$ after his proposal response.

Negotiations among adaptive agents are guaranteed to terminate.

3 ASP-Prolog

The ASP-Prolog system [8] represents an extension of a modular Prolog system which enables the integration of Prolog-style reasoning with Answer Set Programming (ASP). The first implementation of ASP-Prolog dates back to 2004 [2], and we have recently embarked in a redesign and re-implementation of the system using more modern Prolog and ASP technology, as discussed in [8].

An ASP-Prolog program is composed of a hierarchy of modules, where each module can be declared to contain either Prolog code or ASP code. Each module provides an interface which enables to export predicate declarations and import declarations from other modules. In the current implementation, the root of the module hierarchy is expected to be a Prolog module, that can be interacted with using the traditional Prolog-style query-answering mechanism.

Each module is provided with the ability to access the intended models of other modules; in particular, the intended models of each module (i.e., the least Herbrand model of a Prolog module and the answer sets of an ASP module) are themselves automatically realized as individual modules, that can be queried. The interactions among modules can be realized using the following built-in constructs:

- m : `model`(X) succeeds if X is the name of a module representing one of the intended models of the module m—i.e., a module representing the least Herbrand model (resp. an answer set) of a Prolog (resp. ASP) module m.
- m : p succeeds if the atom p is entailed in all the intended models of the module m; in particular, if m represents an answer set of another module, then this test will simply verify whether p is entailed by that particular answer set.
- The content of modules can be retrieved (using the predicate m : `clause`(R)) and modified by other modules. The predicates `assert` and `retract` can be applied to add or remove clauses from a module.

Let us assume we have one Prolog module $\mathtt{p}_1$ and two ASP modules $\mathtt{q}_1$ and $\mathtt{q}_2$:

$\mathbf{p}_1$	$\mathbf{q}_1$	$\mathbf{q}_2$
	`:- asp_module.`	`:- asp_module.`
`:- use_module(q1).`		`:- use_module(q1).`
`:- use_module(q2).`		
`p :- q1:r.`	`r :- not s.`	`a :- not s.`
`s :- q2:a.`	`h.`	`s :- not a.`
`t :- q2:model(X), X:a.`		`q :- q1:h.`
`v :- q1:assert(s), q1:r`		

The queries $\mathtt{p}_1 : \mathtt{p}$, $\mathtt{p}_1 : \mathtt{t}$, and $\mathtt{q}_1 : \mathtt{q}$ are successful, while $\mathtt{p}_1 : \mathtt{s}$ fails. On the other hand, the execution of $\mathtt{p}_1 : \mathtt{v}$ would fail, adding the fact `s` to the module $\mathtt{q}_1$.

4 A Platform for Negotiation Systems

In this section, we describe an ASP-Prolog based platform for the development of negotiation agents which employ ALD in their negotiation. The organization of the platform is illustrated in Figure 1. At the lowest level, we develop an ASP-Prolog layer that provides an implementation of abductive logic programming. This is used, in turn, to enable the representation of n-KBs and to handle the basic handling of negotiation proposals. At the top level, we have Prolog programs that can interact and coordinate the execution of agents performing negotiation. The layers of this architecture are discussed in the following subsections.

4.1 ALP Modules

Since the semantics of ALP and ALP with preferences is defined by answer sets of extended logic programs, it is natural to view ALPs as another type of modules that can be accessed and used in the same way as other modules in ASP-Prolog. To achieve this goal, we extend ASP-Prolog with the following predicates:

- `use_alp(+Name,` $+P^r$`,` $+P^a$`, +[Options])`: this predicate has the same effect as the predicate `use_module(`P`)` for a Prolog or an ASP-module.

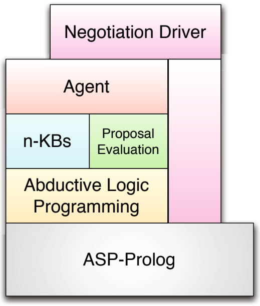


Fig. 1. Negotiation Architecture

It compiles the program (P^r, P^a) with the corresponding options specified in the list $Options$ into a new ASP module, $Name$. The compilation process is illustrated in Algorithm 1. Similarly to the case of ASP modules, the ALP belief sets of (P^r, P^a) are created as (sub)modules of the module produced by the algorithm. Accessing a belief set of the program or its content is done in the same way as for ASP modules. Some shorthands are also provided, e.g., `use_alp(+`P^r`, +`P^a`)` where the module name is specified as P^r.

• `most_preferred(+Name, ?M)`: this predicate allows users to query or check for a most preferred belief set of the ALP program referred to by $Name$.

• `more_preferred(+Name, ?M`$_1$`, ?M`$_2$`)`: this predicate allows users to compare two belief sets w.r.t. the $prefer$ relation.

Algorithm 1. ALP Compile

```
Require: An abductive logic program $P = (P^r, P^a)$; a name $Name$
Ensure: An ASP module $Name$.
  Compile $P^a$ program into a regular answer set program $P^a_1$;
    ∘ introduce the atom $ok(n)$ in the body of rule $n$;
    ∘ generate choice rules of the form $0\{ok(n)\}1.$ for each rule $n$ in $P^a$.
  Generate an ASP module containing the rules $P' = P^r \cup P^a_1$.
```

Example 9. Consider the ALP $P = (P^r, P^a)$ with preference:

$$\begin{aligned} P^r &= \{\, prefer(n_1, n_2), \quad s \leftarrow, \quad \leftarrow not\, p,\ not\, q \,\}, \\ P^a &= \{\, n_1 :\ p \leftarrow not\, r, \quad n_2 :\ q \leftarrow not\, r \,\}. \end{aligned}$$

It is easy to see that P has three belief sets $A_1 = \{p, s, prefer(n_1, n_2)\}$, $A_2 = \{q, s, prefer(n_1, n_2)\}$, and $A_3 = \{p, q, s, prefer(n_1, n_2)\}$ which are belief sets of $P^r \cup \{n_1\}$, $P^r \cup \{n_2\}$, $P^r \cup \{n_1, n_2\}$, respectively[3]. Suppose that P^r and P^a are stored in the files `pr.lp` and `pa.lp` respectively. The command `?- use_alp(p, pr, pa)` compiles $P = (P^r, P^a)$ into the module `p`. It also creates three submodules `p1`, `p2`, and `p3` which correspond to A_1, A_2, and A_3, respectively. The next table displays some queries to p and the corresponding answers.

Query	Answer
`?- p:model(X).`	`X = p1; X = p2; X = p3`
`?- findall(E,(current_predicate(p:P/A), functor(E,P,A),p:E),L).`	`L = [s,prefer(n1,n2)]`
`?- p:model(X), X:p.`	`X = p1; X = p3`
`?- most_preferred(p, X).`	`X = p1`

The first query asks for belief sets of `p`. The second query asks for all atoms that are true in `p`, i.e., those belonging to all belief sets of P. The third query identifies models in which p (the atom) is true. The last query asks for a most preferred belief set of `p`. □

[3] Rule labels refer to the rules in P^a.

4.2 Negotiation Agent

The n-KB layer of Fig. 1 allows the high-level description of the n-KB of a negotiation agent and its mapping to the corresponding ALP module. A negotiation agent is described by a n-KB $K = \langle P, L, B, H, N^{\prec} \rangle$, where $P = (P^r, P^a)$, (L, B) is a dis-information w.r.t. P, H is set of assumptions, and $N^{\prec}$ is set of negotiated conditions. For convenience, we introduce a simple specification language for agents as shown in Table 1 (left). The complete description of a n-KB is provided in a file, with different sections corresponding to the different components of the n-KB.

An agent is compiled into ALP (and, in turn, into ASP as discussed in the previous section) by the command `load_agent(+Agent, +File, +[Options])`, where $Agent$ is the name of the agent, $File$ is file encoding the agent specification, and the list of options $[Options]$ for use with the program $P = (P^r, P^a)$. The command will compile the n-KB into a module, named $Agent$, whose submodules correspond to belief sets of the ALD-program $\delta(P, L, B)$. This allows the users to access belief sets, compute most preferred belief sets, as well as compare belief sets of $\delta(P, L, B)$. The overall structure of the compiler is sketched in Algorithm 2.

Example 10. Consider the n-KB K_s in Ex. 4. Suppose that the n-KB is stored in `seller.lp`. Its description is shown on the right in Table 1. The query `?-load_agent(s,seller)` will compile the n-KB into an ALP module named `s`, denoting the agent s. The module has submodules corresponding to the belief sets of $\delta(P_s, L_s, B_s)$ (Ex. 3) whose content can be accessed using the ASP-Prolog module interface discussed earlier. □

Table 1. Agent Specification

Syntax	Example: Seller agent	
`declare_pr:`	`declare_pr:`	
The program P^r	The program P_s^r (Example 2)	
`declare_pa:`	`declare_pa:`	
The program P^a	The program P_s^a (Example 2)	
`declare_lying:`	`declare_lying:`	
Literals in L	$quality_B.$	
`declare_bs:`	`declare_bs:`	
Literals in B	$quality_A.$	
`declare_hypotheses:`	`declare_hypotheses:`	
Literals in H	$age \geq 65.$	$student.$
	$cash.$	$mail\_list$
`declare_goal:`	`declare_goal:`	
Literals in N and	$high\_pr.$	
the preference order $\prec$	$low\_pr.$	$lowest\_pr.$
	$prefer(low\_pr, lowest\_pr).$	
	$prefer(high\_pr, low\_pr).$	
	$prefer(high\_pr, lowest\_pr).$	

Algorithm 2. n-KB Compilation

```
Require: An Agent name
Require: A n-KB $K = \langle P, L, B, H, N^{\prec} \rangle$ where $P = (P^r, P^a)$
Ensure: An ALP module representing $\delta(P, L, B)$.
  Compute $I = \{ r \mid r \in P^r$ and $head(r) \cap L^{\neg} \neq \emptyset \}$
  $P_1^a = P^a \cup I \cup L \cup B$
  Assign labels to the rules in $P_1^a$
  Compute
      $\Phi = \{ prefer(n_i, n_j) \mid n_i \in I$ and $n_j \in P^a \cup H \} \cup$
          $\{ prefer(n_j, n_t) \mid n_j \in P^a \cup H \cup I$ and $n_t \in (L \cup B) \}$
  $P_1^r = P^r \setminus I \cup \Phi$
  Compile the program $P_1 = (P_1^r, P_1^a)$ using the use_alp predicate.
```

4.3 Computing and Evaluating Proposals

The proposal evaluation layer (Fig. 1) provides a collection of predicates to generate general proposals and to evaluate proposals for acceptability, rejectability, and negotiability. Let us discuss the key predicates; all examples refer to s and b as the seller and buyer whose n-KBs are specified in Examples 4 and 5. We assume that the n-KBs are already compiled into modules s and b, respectively, using the `load_agent` command.

- `proposal(+Agent,[?G,?S,?R])`: this predicate succeeds if $\langle G, S, R \rangle$ is a proposal for the agent $Agent$. Note that this predicate can be used to generate as well as test a proposal. For example, the query `?- proposal(s,[G,S,R])` generates an arbitrary proposal that the seller agent `s` can create, given her n-KB (Ex. 10); one possible answer is `G = [`low_pr`], S = [], R = [`$product_A$`]`.
- `proposal(+Agent,?M,[?G,?S,?R])`: $\langle G, S, R \rangle$ is a proposal for the agent $Agent$ with supporting belief set M. For example, assume that the module s has a submodule $s1$ containing low_pr, $student$, and $product_B$; the query

  `?- proposal(s,s1,[G,S,R]).`

  asks for a possible proposal supported by the belief set described by module $s1$; it returns the answer `G = [`low_pr`], S = [`$student$`], R = [`$product_B$`]`. A most preferred proposal for s can be obtained by the query:

  `?- most_preferred(s, M), proposal(s,M,[G,S,R]).`
- `acceptable(+Agent,[+G,+S,+R])`: $\langle G, S, R \rangle$ is an acceptable proposal for the agent $Agent$. For example, the query

  `?- acceptable(s, [[`$high\_pr$`],[`$product_A$`], []]).`

  asks if $\langle high\_pr, \{product_A\}, \emptyset \rangle$ is acceptable for agent s; this query succeeds.
- `negotiable(+Agent,[+G,+S,+R])`: $\langle G, S, R \rangle$ is a negotiable proposal for the agent $Agent$. The query

  `?- negotiable(s, [[`$high\_pr$`],[`$product_A$`, `$quality_A$`], []]).`

  checks whether the proposal $\langle high\_pr, \{product_A, quality_A\}, \emptyset \rangle$ is negotiable for the agent s, and it succeeds.
- `rejectable(+Agent,[+G,+S,+R])`: $\langle G, S, R \rangle$ is a rejectable proposal for the agent $Agent$. The query

  `?- rejectable(b, [[`$high\_pr$`],[`$product_A$`, `$quality_A$`, `$maker_C$`], []]).`

checks if $\langle high\_pr, \{product_A, quality_A, maker_C\}, \emptyset\rangle$ is rejectable for agent b; it succeeds, since this proposal is neither acceptable nor negotiable for b.

4.4 Agents: Responses, Strategies, and Negotiations

The design of a negotiation agent requires, in addition to its knowledge, the ability of applying strategies for the generation of proposals (e.g., act as an adaptive agent) and for the selection of responses (e.g., develop conscious responses). To this end, our negotiation architecture provides predicates for computing conscious responses and updates of agents. We next describe these predicates. Other strategies can be realized (as relatively simple Prolog modules) and their development is part of the future work.

- `response(+Agent,[?G2,?S2,?R2],[+G1,+S1,+R1])`: $\langle G2, S2, R2\rangle$ is a conscious response (by $Agent$) to the proposal $\langle G1, S1, R1\rangle$. Note that the predicate can be used to check responses (for consciousness) or to generate conscious responses. For example, the answer to the query
  `?- response(b,[G,S,R],[[low_pr],[cash],[product`$_A$`, quality`$_A$`, maker`$_C$`]])`
  which asks for a conscious response by b to the proposal
  $\langle$ low_pr, $\{cash\}$, $\{product_A$, $quality_A$, $maker_C\}$ $\rangle$ is
  `G=[`low_pr`], S=[`$product_A$`, `$quality_A$`, `$maker_C$`], R = [`$cash$`]`
- `update_kb(+Agent, +Type, +Flag,+Value)`: where $Type$ is one of {`pr`, `pa`, `lying`, `bs`, `hypotheses`, `goal`} and $Flag$ is either `true` or `false`. $Type$ specifies the part of the agent's KB, that needs to be updated. $Flag$ is true (false) indicating whether the content specified in $Value$ has to be added or removed ($Value$ could be either a constant or a list of constants). This predicate updates the `declare_type` part of the agent. For instance, the query
  `?- update_kb(s, pr, true, `$student$`)`
  updates the program P^r of the seller n-KB with the fact $student$, i.e., adding the fact $student$ to the program P^r. Intuitively, this query should be executed after the seller learns that the buyer is a student. On the other hand, the query
  `?- update_kb(s, hypotheses, false, `$student$`)`
  updates the set of hypothesis H_s of the seller by removing $student$ from H_s, e.g., after she learns that the buyer is indeed a student.

4.5 A Program for Automated Negotiation

The top-level of the proposed framework is a *coordinator agent*; the role of the coordinator is to control the exchanges between the negotiating agents, enforcing the proper ordering in the exchanges and acting as a communication channel among the negotiating agents. The coordinator agent is a relatively simple Prolog module. The entry point of the coordinating agent is the predicate `main`, which receives in input the names of the two negotiating agents and the files containing their n-KBs:

```
main(Name1, Agent1, Name2, Agent2, [G,S,R]) :-
   load_agent(Name1, Agent1), load_agent(Name2, Agent2),
   negotiation(Name1, Name2, [G,S,R]).
```

The negotiation starts with an agent identifying a most preferred proposal and proposing it to the other agent. The proposal identified by $[G, S, R]$ is the final outcome of a negotiation. The actual negotiation process is an iterative process (implemented by the predicate `round`), which alternates generation of responses between the two agents. In each round, the receiving agent computes a counter proposal and updates her n-KB. The coordinator agent updates the history of the negotiation to ensure that agents do not repeat their answers. Multiple traces can be obtained by asking for different answers.

```
1: negotiation(A, B, [G1,S1,R1]):-
2:   most_preferred(A,M), proposal(A,M,[G,S,R]), print_prop(A,G,S,R),
3:   round(A,B,[G,S,R],[(A,[],[G,S,R])], [G1,S1,R1]).
4: round(A, B, [G,S,R],History,[G,S,R]):-  print_prop(A,G,S,R),
5:    response(B,[G1,S1,R1],[G,S,R]),
6:    check_repeated((B,[G,S,R],[G1,S1,R1]),History),
7:    ([G1,S1,R1]==[true,[],[]] -> write('Accepted') ;
8:    ([G1,S1,R1]==[false,[],[]]-> write('Rejected') ;
9:    append([(B,[G,S,R],[G1,S1,R1])],History,History1),
10:   agent(B, Hyp, _, _, _), intersection(Hyp, R, SH1),
11:   negated(Hyp, HypN), intersection(HypN, R, SH2),
12:   union(SH1, SH2, SH), update_KB(B, pr, true, SH),
13:   round(B, A, [G1,S1,R1],History1,[G1,S1,R1])        )).
```

where `agent(·)` provides the components of the agent and `negated`(S, S') is true for $S' = S^{\neg}$. The subgoals in lines 7–12 are used to update the agent (see definition of adaptive agent), while line 6 avoids repetitions of proposals.

Example 11. The following is an example of some negotiations when running the query
`?- main(s, seller, b, buyer, [G,S,R]).`

```
Agent s:  proposal {[high_pr],[],[]}
G=[high_pr], S=[], R=[]
Rejected yes ? ;
Agent b: proposal {[low_pr],[qualityB,makerC,productB],[]}
Agent s: proposal {[low_pr],[cash],[-(qualityB)]}
Agent b: proposal {[low_pr],[qualityA,makerC,productA],[]}
Agent s: proposal {[low_pr],[cash],[]}
Agent b: proposal {[low_pr],[qualityA,makerC,productA],[cash]}
Accepted yes ?
```

The seller starts with $\langle\{high\_pr\}, \emptyset, \emptyset\rangle$, which the buyer rejects. The buyer proposes $\langle\{low\_pr\}, \{quality_B, maker_C, product_B\}, \emptyset\rangle$. This does not work for the seller, as he does not want to lie about $quality_B$. The buyer proposes the alternative $\langle\{low\_pr\}, \{quality_A, maker_C, product_A\}, \emptyset\rangle$ for which the seller wants cash. The buyer agrees and the seller agrees to sell the product, with BS about the quality of product A. □

5 Conclusion

In this paper, we introduced the design of a logic programming platform to implement negotiating agents. The model of negotiation supported by the proposed platform enables the representation of agents with incomplete knowledge and capable of choosing dishonest answers in building their offers and counter-offers. The architecture has been entirely developed using the ASP-Prolog system, taking advantage of the ability of combining Prolog and ASP modules. We believe this architecture is quite unique in the level of flexibility provided and in its ability to support easy extensions to capture, e.g., different agent strategies and behaviors.

We are currently extending the architecture to provide several built-in agent strategies (e.g., to represent agents with different levels of dishonesty), allow actual concurrency in the agents' interactions (e.g., through a Linda-style blackboard), and implementing real-world scenarios.

References

1. Chen, W., Zhang, M., Foo, N.: Repeated negotiation of logic programs. In: Proc. 7th Workshop on Nonmonotonic Reasoning, Action and Change (2006)
2. Elkhatib, O., Pontelli, E., Son, T.C.: $\mathbb{ASP} - \mathbb{PROLOG}$: A System for Reasoning about Answer Set Programs in Prolog. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 148–162. Springer, Heidelberg (2004)
3. Fatima, S.S., Wooldridge, M., Jennings, N.R.: Bargaining with incomplete information. Ann. Math. Artif. Intell. 44(3), 207–232 (2005)
4. Frankfurt, H.G.: On Bullshit. Princeton Univ. Press, Princeton (2005)
5. Kakas, A.C., et al.: Agent planning, negotiation and control of operation. In: ECAI (2004)
6. Kraus, S.: Negotiation and cooperation in multi-agent environments. AIJ 94(1-2) (1997)
7. Mahon, J.E.: Two definitions of lying. J. Applied Philosophy 22(2), 211–230 (2008)
8. Pontelli, E., Son, T.C., Nguyen, N.-H.: Combining answer set programming and prolog: The ASP–PROLOG system. In: Tran, S. (ed.) Gelfond Festschrift. LNCS (LNAI), vol. 6565, pp. 452–472. Springer, Heidelberg (2011)
9. Sadri, F., Toni, F., Torroni, P.: An abductive logic programming architecture for negotiating agents. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 419–431. Springer, Heidelberg (2002)
10. Sakama, C., Inoue, K.: Prioritized Logic Programming and its Application to Commonsense Reasoning. Artificial Intelligence 123(1-2), 185–222 (2000)
11. Sakama, C., Inoue, K.: Negotiation by abduction and relaxation. In: AAMAS, pp. 1018–1025. ACM Press, New York (2007)
12. Sakama, C., Son, T.C., Pontelli, E.: A logical formulation for negotiation among dishonest agents (2010), www.cs.nmsu.edu/~tson/papers/neg2010.pdf
13. Son, T.C., Sakama, C.: Negotiation using logic programming with consistency restoring rules. In: IJCAI, pp. 930–935 (2009)
14. Wooldridge, M., Parsons, S.: Languages for negotiation. In: ECAI (2000)
15. Zlotkin, G., Rosenschein, J.S.: Incomplete information and deception in multi-agent negotiation. In: IJCAI, pp. 225–231 (1991)

Advances in *gringo* Series 3

Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub[⋆]

Institut für Informatik, Universität Potsdam

Abstract. We describe the major new features emerging from a significant redesign of the grounder *gringo*, building upon a grounding algorithm based on semi-naive database evaluation. Unlike previous versions, rules only need to be *safe* rather than domain-restricted.

1 Introduction

A distinguishing feature of Answer Set Programming (ASP; [1]) is its highly declarative modeling language along with its domain-independent grounding systems, like *lparse* [2], *dlv* [3], and *gringo* [4]. This paper is dedicated to the features of the new major release of *gringo*, starting with version 3. Most notably, this series only stipulates rules to be *safe* (cf. [5]) rather than λ-restricted [6], as in previous versions of *gringo*. Hence, programs are no longer subject to any restriction guaranteeing a finite grounding. Rather, this responsibility is left with the user in order to provide her with the greatest flexibility. This general setting is supported by a grounding algorithm based on semi-naive database evaluation (cf. [5]), closely related to that of *dlv*. In what follows, we elaborate upon the new features emerging from this significant redesign. For a thorough introduction of *gringo*'s language, please consult the manual available at [7].

2 Advances in *gringo* Series 3

The most significant change from *gringo* 2 to 3 is that the basic grounding procedure of *gringo* 3 no longer instantiates the rules of a logic program strictly along a predefined order. This enables more convenient predicate definitions in terms of (positive) recursion. E.g., consider a λ-restricted "connected graph design" program:

```
 node(1..5).
{ edge(1,X) } :- node(X).
{ edge(X,Y) } :- reached(X), node(Y).
 reached(Y)    :- edge(X,Y),  node(X;Y).
               :- node(X),    not reached(X).
```

In *gringo* 3, `reached/1` can be defined more conveniently as follows:

```
 reached(Y)    :- edge(X,Y).
```

[⋆] Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

In fact, additional domain information via `node/1`, needed in rule-wise grounding to "break" the cyclic definition of `edge/2` and `reached/1`, is not anymore required in view of semi-naive evaluation.

Since *gringo* features built-in arithmetics and uninterpreted functions, it deals with potentially infinite Herbrand universes, and the safeness condition does not guarantee termination. For instance, *gringo* 3 does not terminate on the following program:

```
succ(X,X+1) :- succ(X-1,X).  succ(X,X+1) :- zero(X).  zero(0).
```

In fact, the program has an infinite answer set, and it is not λ-restricted in view of the first (recursive) rule. Although it may appear disturbing that the termination of *gringo* 3 is not always guaranteed, we deliberately chose to not reintroduce any syntactic finiteness check, and rather leave it to the user to include appropriate stop conditions limiting the "relevant" ground instances of rules. E.g., one may replace the first rule by:

```
succ(X,X+1) :- succ(X-1,X), X < 42.
```

Since *gringo* 3 evaluates built-ins while instantiating a rule, it stops grounding at `succ(41,42)`, so that only finitely many relevant ground rules are produced.

The design decision to not enforce (syntactic) conditions guaranteeing termination gives users the freedom to write "clever" encodings, where syntactic checks are bound to fail. To see this, consider the following encoding of a universal Turing Machine:

```
tm(S, L,A, R)         :- init(S),            tape(L,A,R).
tm(SN,L,AL,r(AN,R))   :- tm(S,l(L,AL),A,R),  d(S,A,AN,SN,l).
tm(SN,n,0, r(AN,R))   :- tm(S,n,A,R),        d(S,A,AN,SN,l).
tm(SN,l(L,AN),AR,R)   :- tm(S,L,A,r(AR,R)),  d(S,A,AN,SN,r).
tm(SN,l(L,AN),0, n)   :- tm(S,L,A,n),        d(S,A,AN,SN,r).
```

The idea is to represent configurations of the universal Turing Machine by instances of `tm(State,LTape,Symbol,RTape)`, where `State` is a state of the machine that is run, `Symbol` is the tape contents at the current read/write-head position, and `LTape` and `RTape` are (usually) functions representing the tape contents to the left and right, respectively, of the current position. Starting from an initial state and tape, successor configurations are calculated relative to a transition table given by instances of `d(State,Symbol,NSymbol,NState,Direction)`. For instance, if the direction is `l` for "left," the contents of `NSymbol` is appended to the tape contents to the right, and the next symbol to the left is taken as the contents at the new position, while also removing it from the left-hand side's tape contents. Hereby, we use `n` to indicate infinitely many blanks `0` to the left or right of the current position, and dedicated rules take care of "generating" a blank on demand when a tape position is visited first. A machine to run, e.g., a 3-state Busy Beaver machine, can then be given by facts like:

```
d(a,0,1,b,r).  d(b,0,1,a,l).  d(c,0,1,b,l).  init(a).
d(a,1,1,c,l).  d(b,1,1,b,r).  d(c,1,1,h,r).  tape(n,0,n).
```

If we run *gringo* 3 on the universal Turing Machine encoding along with the facts specifying the 3-state Busy Beaver machine, it generates all traversed configurations, where the final one is as follows:

```
tm(h,l(l(l(l(n,1),1),1),1),1,r(1,n))
```

The fact that *gringo* 3 terminates tells us that the 3-state Busy Beaver machine halts after writing six times the symbol `1` to the tape. However, given that *gringo* 3 can be used to simulate any machine and the halting problem, in general, is undecidable, it is also undecidable whether semi-naive evaluation yields a finite grounding, i.e., whether *gringo* 3 eventually terminates.

The language of *gringo* 2 [4] already included a number of **aggregates**, most of which are still supported by *gringo* 3: `#count`, `#sum`, `#min`, `#max`, `#avg`, `#even`, and `#odd`[1]. The support also includes backward compatibility to the traditional notation of *cardinality* and *weight constraints* [2], `l{...}u` and `l[...]u`, respectively, rather than `l#count{...}u` and `l#sum[...]u`. As with *gringo* 2, the condition connective '`:`' allows for qualifying local variables within an aggregate (or variable-sized conjunctions/disjunctions). Hence, *gringo* 2 and 3 both accept the next program:

```
d(1;2;3).  { p(X) : d(X) }.
all :- S = #sum[ d(X) : d(X) = X ], S #sum[ p(X) : d(X) = X ].
```

Note that all local variables (named `X`) within aggregates are bound via a domain predicate [2], `d/1`, on the right-hand side of '`:`'. While such binding via domain predicates (or built-ins) had been mandatory with *gringo* 2, it can often be omitted with *gringo* 3. E.g., the last rule above can also be written shorter as follows:

```
all :- S = #sum[ d(X) = X ], S #sum[ p(X) = X ].
```

After investigating the rules with atoms of the predicates `d/1` and `p/1` in the head and collecting their ground instances, *gringo* 3 notices that no further rule can derive any instances, so that both domains are limited to `1`, `2`, and `3`. Hence, explicit domain information is not needed to identify all eligible values for the local variables `X` in the remaining rule. In fact, since `d/1` is a domain predicate, *gringo* 3 (deterministically) calculates `S = 6`, which is then taken as the lower bound in `6 #sum[p(1),p(2),p(3)]`. However, note that a similar omission of '`: d(X)`' is not admissible in the head of the second rule above, since it would violate the safeness requirement. Finally, *gringo* 3 does currently not support implicit domains of local variables if an aggregate is involved in (positive) recursion; e.g., the following modified rule is safe, but not yet supported:

```
p(S) :- S = #sum[ d(X) = X ], S #sum[ p(X) = X ].
```

Implicit domains in recursive aggregates are subject to future work (cf. Section 3).

Optimization statements, which can be specified via the directives `#minimize` and `#maximize`, are syntactically very similar to aggregates, and *gringo* 3 fully supports implicit domains for local variables in them. (Since optimization statements are not part of logic program rules, they cannot be "applied" to derive any atom.) With *lparse* and *gringo* 2, it is possible to provide multiple optimization statements with implicit priorities depending on their order in the input: by convention [2], the last statement is more significant than the second last one, which in turn is more significant then the one before, etc. This convention is also adopted by *gringo* 3, which must be taken into account when writing a sequence of optimization statements like the following one:

```
#minimize[ p(X) = X ].
#maximize[ p(X) = X ].
```

[1] The `#times` aggregate is currently not supported by *gringo* 3; while it requires an involved compilation to "Smodels Internal Format" [2], we are not aware of any application using it.

According to the order, the `#maximize` objective takes precedence over `#minimize`. Since such implicit prioritization necessitates a lot of care to be used properly and also undermines the idea of declarative programming, *gringo* 3 supports explicit priorities via *precedence levels*, provided via the connective '`@`'. This can be used to override default prioritization, e.g., as follows:

```
#minimize[ p(X) = X @  X ].
#maximize[ p(X) = X @ -X ].
```

If we assume the domain of `p/1` to contain the values `1`, `2`, and `3`, the corresponding ground optimization statements include the following weighted literals (sorted by their precedence levels):

```
#minimize[ p(3) = 3 @  3, p(2) = 2 @  2, p(1) = 1 @  1 ].
#maximize[ p(1) = 1 @ -1, p(2) = 2 @ -2, p(3) = 3 @ -3 ].
```

These optimization statements involve six precedence levels, and a greater level is more significant than a smaller one. Accordingly, our main objective is `p(3)` to be false, followed by `p(2)`, and then `p(1)`. Furthermore, the three negative levels (which are superseded by the positive ones that are greater) express that we would also like `p(1)` to be true, then `p(2)`, and finally `p(3)`. Observe that the optimization priorities are fully determined by precedence levels (and weights), so that there are no implicit priorities based on ordering anymore. We note that prioritization of optimization objectives via precedence levels and weights is also supported by *dlv*, which offers weak constraints [3]. In fact, extending *gringo* by weak constraints is subject to future work.

For selectively **displaying** atoms, *lparse* and *gringo* 2 support `#hide` and `#show` statements to, at the predicate level, decide which atoms in an answer set ought to be presented or suppressed, respectively. Albeit such output restriction mainly serves user convenience, there are also profound application scenarios, such as the enumeration of projected answer sets [8] offered by *clasp* (option `--project`). In fact, the following methodology had been used to, at the predicate level, project Hamiltonian cycles in a clumpy graph down to edges in an underlying "master graph":

```
% Derive "mc" from "hc"
mc(C1,C2) :- hc(C1,V1,C2,V2), C1 != C2.
% Output PROJECTION to "mc"
#hide.
#show mc(C1,C2).
```

To support *output projection* not only at the predicate but also *at the atom level*, *gringo* 3 allows for conditions, connective '`:`' followed by domain predicates and/or built-ins, within `#hide` and `#show` statements. Given this, defining a predicate `mc/2` can be omitted and output projection be accomplished more conveniently as follows:

```
% Output PROJECTION
#hide.
#show hc(C1,V1,C2,V2) : C1 != C2.
```

As with precedence levels for optimization, the possibility to distinguish outputs qualified via `#hide` and `#show` at the atom level, rather than at the level of predicates, contributes to declarativeness, as it abolishes the need to define auxiliary predicates within a logic program only for the sake of projecting the displayed output to them.

A number of **built-in** (arithmetic) comparison **predicates**, viz., '`==`', '`!=`', '`<=`', '`>=`', '`<`', '`>`', and (variable) assignments, via '`=`', were already included in the input language of *gringo* 2. In *gringo* 3, respective comparison predicates are generalized to *term comparisons*, that is, they do not anymore raise an error like "comparing different types," as encountered with (some versions of) *gringo* 2 when writing, e.g., `2 < f(a)` or alike. Furthermore, while the left-hand side of '`=`' must be a variable, the *generalized assignment operator* '`:=`' offered by *gringo* 3 admits composite terms (including variables) on its left-hand side. Hence, it is possible to simultaneously assign multiple variables in a rule like the following one:

```
p(X,Y,Z) :- (X,f(Y,a,Z)) := (a,f(b,a,c)).
```

As with '`=`', we still require a right-hand side of '`:=`' to be instantiable before '`:=`' is evaluated. E.g., a rule like the following one is currently not supported by *gringo*:

```
p(X) :- (X,a) := (a,X).
```

Sometimes, the built-ins offered by a grounder may be too spartan to accomplish sophisticated calculations, and encoding them may likewise be involved and possibly too space-consuming. To nonetheless allow users to accomplish application-specific calculations during grounding, *gringo* 3 comes along with an **embedded scripting language**, viz., *lua* [9]. For instance, the greatest common divisor of numbers given by instances of a predicate `p/1` can be calculated via *lua* and then be "saved" in the third argument of a predicate `q/3`, as done in the following program:

```
#begin_lua
  function gcd(a,b)
    if a == 0 then return b else return gcd(b % a,a) end
  end
#end_lua.

q(X,Y,@gcd(X,Y)) :- p(X;Y), X < Y.  p(2*3*5;2*3*7;2*5*7).
```

When passing this program to *gringo* 3, it for one calculates the numbers being arguments of predicate `p/1`, `30`, `42`, and `70`, while the implementation of the `gcd` function in *lua* is used to derive the following facts over predicate `q/3`:

```
q(30,42,6).  q(30,70,10).  q(42,70,14).
```

Beyond sophisticated arithmetics, *lua* also allows for environment interaction. E.g., it provides interfaces to read off values from a database. In the following example, we use `sqlite3`, embedded into the precompiled *gringo* 3 binaries available at [7]:

```
#begin_lua
  local env  = luasql.sqlite3()
  local conn = env:connect("db.sqlite3")
  function query()
    local cur = conn:execute("SELECT * FROM test")
    local res = {}
    while true do
      local row = {}
      row = cur:fetch(row,"n")
      if row == nil then break end
      res[#res + 1] = Val.new(Val.FUNC,row)
    end
    cur:close()
    return res
  end
#end_lua.

p(X,Y) :- (X,Y) := @query().
```

Here, a *lua* function `query` is used to read data from a table called `test`. Although we do here not delve into the details of *lua*, there is one line that deserves attention:

```
res[#res + 1] = Val.new(Val.FUNC,row)
```

If `test` contains the tuples ⟨1,a⟩, ⟨2,b⟩, and ⟨3,c⟩, they are successively inserted into the array `res`. The collected tuples are then taken to construct the following facts:

```
p("1","a").  p("2","b").  p("3","c").
```

We note that the generation of terms via *lua* is similar to "value invention" [10], allowing for custom built-in predicates evaluated during grounding.

3 Discussion

The redesign of *gringo* is an important step in consolidating the distinct grounding approaches originated by *lparse* and *dlv*. A common ASP language unifying the constructs of both approaches is already envisaged as a joint effort of the teams at Calabria and Potsdam. Although *dlv* and *gringo* now share many commonalities, like safety, semi-naive database evaluation, function symbols, and Turing completeness, they still differ in aspects like finiteness criteria, indexing, connectivity, incrementality, recursive aggregates, backtracking and -jumping. Hence, it is interesting future work to further investigate the different designs and consolidate them wherever possible.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Syrjänen, T.: Lparse 1.0 user's manual, `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL 7(3), 499–562 (2006)
4. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder gringo. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 502–508. Springer, Heidelberg (2009)
5. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)

6. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
7. http://potassco.sourceforge.net
8. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected Boolean search problems. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 71–86. Springer, Heidelberg (2009)
9. Ierusalimschy, R.: Programming in Lua (2006), http://www.lua.org
10. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. AMAI 50(3-4), 333–361 (2007)

A Portfolio Solver for Answer Set Programming: Preliminary Report

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub*,
Marius Thomas Schneider, and Stefan Ziller

Institut für Informatik, Universität Potsdam

Abstract. We propose a portfolio-based solving approach to Answer Set Programming (ASP). Our approach is homogeneous in considering several configurations of the ASP solver *clasp*. The selection among the configurations is realized via Support Vector Regression. The resulting portfolio-based solver *claspfolio* regularly outperforms *clasp*'s default configuration as well as manual tuning.

1 Introduction

Answer Set Programming (ASP; [1]) has become a prime paradigm for declarative problem solving due to its combination of an easy yet expressive modeling language with high-performance Boolean constraint solving technology. In fact, modern ASP solvers like *clasp* [2] match the performance of state-of-art satisfiability (SAT) checkers, as demonstrated during the last SAT competition in 2009. Unfortunately, there is a price to pay: despite its theoretical power [3], modern Boolean constraint solving is highly sensitive to parameter configuration. In fact, we are unaware of any true application on which *clasp* is run in its default settings. Rather, in applications, "black magic" is used to find suitable search parameters. Although this is well-known and also exploited in the SAT community, it is hardly acceptable in an ASP setting for the sake of declarativity. The most prominent approach addressing this problem in SAT is *satzilla* [4], aiming at selecting the most appropriate solver for a problem at hand.

Inspired by *satzilla*, we address the lack of declarativity in ASP solving by exploring a portfolio-based approach. To this end, we concentrate on the solver *clasp* and map a collection of instance features onto an element of a portfolio of distinct *clasp* configurations. This mapping is realized by appeal to Support Vector Regression [5]. In what follows, we describe the approach and architecture of the resulting *claspfolio* system. We further provide an empirical analysis contrasting *claspfolio*'s performance with that of *clasp*'s default setting as well as the manually tuned settings used during the 2009 ASP competition. In addition, we compare the approach of *claspfolio* with *paramils* [6], a tool for parameter optimization based on local search.

2 Architecture

Given a logic program, the goal of *claspfolio* is to automatically select a suitable configuration of the ASP solver *clasp*. In view of the huge configuration space, the attention is

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

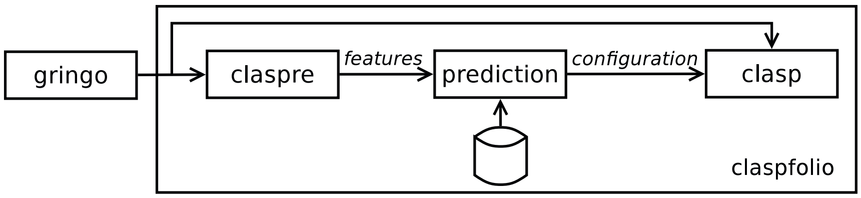


Fig. 1. Architecture of *claspfolio*

limited to some (manually) selected configurations belonging to a portfolio. Each configuration consists of certain *clasp* options, e.g., "*--heuristic=VSIDS --local-restarts*." To approximate the behavior of such a configuration, *claspfolio* applies a model-based approach predicting solving performance from particular features of the input.

As shown in Figure 1, ASP solving with *claspfolio* consists of four parts. First, the ASP grounder *gringo* [7] instantiates a logic program. Then, a light-weight version of *clasp*, called *claspre*, is used to extract features and possibly even solve a (too simple) instance. If the instance was not solved by *claspre*, the extracted features are mapped to a score for each configuration in the portfolio. Finally, *clasp* is run for solving, using the configuration with the highest score. Note that *claspre* and *clasp* use distinct copies of the input (see Figure 1) because preprocessing done by *claspre* may interfere with *clasp* options of configurations in the portfolio.

The features determined by *claspre* can be distinguished into data- and search-oriented ones. The former include 50 properties, such as number of constraints, number or variables, etc. Beyond that, *claspre* performs a limited amount of search to also collect information about solving characteristics. In this way, additional 90 search-related features are extracted, such as average backjump length, length of learned clauses, etc.

Given the features of an instance, *claspfolio* scores each configuration in the portfolio. To this end, it makes use of models generated by means of machine learning during a training phase. In the case of *claspfolio*, we applied Support Vector Regression, as implemented by the `libSVM` package [8]. Upon training, the score $s_k(i)$ of the k-th configuration on the i-th training instance is simply the runtime $t_k(i)$ in relation to the minimum runtime of all configurations in the portfolio: $s_k(i) = \frac{\min_j (t_j(i))}{t_k(i)}$.

A model, i.e., a function mapping instance features to scores, is then generated from the feature-score pairs available for the training set. In production mode, only the features (collected by *claspre*), but not the configurations' scores, are available. Hence, the models are queried to predict the scores of all configurations in the portfolio, among which the one with the highest predicted score is selected for setting up *clasp*.

The portfolio used by *claspfolio* (version 0.8.0) contains 12 *clasp* configurations, included because of their complementary performances on the training set. The options of these configurations mainly configure the preprocessing, the decision heuristic, and the restart policy of *clasp* in different ways. This provides us with a collection of solving strategies that have turned out to be useful on a range of existing benchmarks. In fact, the hope is that some configuration is (a) well-suited for a user's application and (b) automatically selected by *claspfolio* in view of similarities to the training set.

3 Experiments

We conducted experiments on benchmark classes of the 2009 ASP competition [9][1]. All experiments were run on an Intel Xeon E5520 machine, equipped with 2.26 GHz processors and 48 GB RAM, under Linux. The considered systems are *clasp* (1.3.4) and *claspfolio* (0.8.0; based on *clasp* 1.3.4). Runtimes in seconds, per class and in total, are shown in Table 1. The first two columns give benchmark classes along with their numbers of instances (#). The subsequent columns denote particular variants of the considered systems: *clasp* default (*clasp*), *clasp* manually tuned[2] ($clasp^m$), *claspfolio* running a random configuration ($claspfolio^r$), *claspfolio* running the best configuration[3] ($claspfolio^b$), *claspfolio* default (*claspfolio*) as available at [7], and *claspfolio* obtained by cross validation ($claspfolio^v$). The runtime per benchmark instance was limited to $1,200$ seconds, and timeouts are taken as $1,200$ seconds within accumulated results. The third last and the last column ($\times$) in Table 1 provide the speedup of *claspfolio* and $claspfolio^v$, respectively, over *clasp*, i.e., the runtime of *clasp* divided by the one of *claspfolio* or $claspfolio^v$, per benchmark class (and in total in the last row).

The role of $claspfolio^v$ is to evaluate *claspfolio* on unseen instances. We do so by using 10-fold cross validation where the set of all available instances is randomly divided into a training set and a test set, consisting of 90 and 10 percent of the inspected instances, respectively. The regression models generated on the training set are then evaluated on the (unseen) test set. By repeating this procedure ten times, every instance is once solved based on models not trained on the instance.

Comparing *clasp* with $clasp^m$ in Table 1, manual tuning turns out to be mostly successful, and it decreases total runtime roughly by a factor of 3. On two classes, Labyrinth and WireRouting, manual tuning was however counterproductive. This can be explained by the 2009 ASP competition mode, revealing only a few of the available instances per benchmark class during a setup phase, so that the manually tuned parameters may fail on unseen instances. In fact, *claspfolio*, trained on a collection of 3096 instances from the Asparagus benchmark repository[4] and the 2009 ASP competition, turns out to be even more successful in total than $clasp^m$. In particular, it performs better on Labyrinth and WireRouting, where $clasp^m$ failed to improve over *clasp*. Of course, there are also benchmark classes on which manual tuning beats *claspfolio* (most apparently, WeightDomSet), but the fact that *claspfolio* exhibits a total speedup of 3.3 over *clasp* clearly shows the potential of automatic parameter selection. Notably, the total runtime of *claspfolio* exceeds the best possible one, $claspfolio^b$, only by a factor of 1.45, while the expected runtime of a random configuration, $claspfolio^r$, is in total more than a factor of 4 greater than the one of *claspfolio*.

[1] Some too easy/unbalanced classes or instances, respectively, of the competition are omitted. On the other hand, we also ran additional instances for some classes. All instances used in our experiments are available at `http://www.cs.uni-potsdam.de/claspfolio`

[2] The respective parameter settings per benchmark class are reported at `http://dtai.cs.kuleuven.be/events/ASP-competition/Teams/Potassco.shtml`

[3] Note that results of $claspfolio^r$ and $claspfolio^b$ are calculated a posteriori per benchmark instance, using the average or smallest, respectively, runtime of all *clasp* variants in the portfolio.

[4] Available at `http://asparagus.cs.uni-potsdam.de`

Table 1. Runtimes in seconds and speedups on benchmark classes of the 2009 ASP competition

Benchmark Class	#	*clasp*	*clasp*^{m}	*claspfolio*^{r}	*claspfolio*^{b}	*claspfolio*	$\times$	*claspfolio*^{v}	$\times$
15Puzzle	37	510	281	438	111	208	2.4	254	2.0
BlockedNQueens	65	412	374	765	139	264	1.5	410	1.0
ConnectDomSet	21	1,428	54	1,236	30	53	26.9	649	2.2
GraphColouring	23	17,404	5,844	15,304	5,746	5,867	2.9	5,867	2.9
GraphPartitioning	13	135	66	791	57	69	1.9	97	1.4
Hanoi	29	458	130	499	35	175	2.6	233	2.0
Labyrinth	29	1,249	1,728	3.949	112	785	1.5	2,537	0.5
MazeGeneration	28	3,652	569	4,086	558	581	6.2	567	6.4
SchurNumbers	29	726	726	1,193	41	399	1.8	957	0.7
Sokoban	29	18	19	34	12	57	0.3	54	0.3
Solitaire	22	2,494	631	3,569	73	317	7.8	1,610	1.5
WeightDomSet	29	3,572	248	10,091	5	1,147	3.1	5,441	0.6
WireRouting	23	1,223	2,103	1,409	43	144	8.4	289	4.2
Total	377	33,281	12,773	43,364	6,962	10,066	3.3	18,965	1.8

Table 2. Comparison with *paramils* on benchmark classes of the 2009 ASP competition

Benchmark Class	#	*paramils*^{c}	*paramils*^{a}	*claspfolio*	*claspfolio*^{v}	*clasp*	*clasp*^{m}
15Puzzle	37	104	322	208	254	510	281
BlockedNQueens	65	212	352	264	410	412	374
ConnectDomSet	21	28	686	53	649	1,428	54
GraphColouring	23	7,596	10,865	5,867	5,867	17,404	5,844
GraphPartitioning	13	39	86	69	97	135	66
Hanoi	29	35	147	175	233	458	130
Labyrinth	29	462	3,080	785	2,537	1,249	1,728
MazeGeneration	28	700	2,610	581	567	3,652	569
SchurNumbers	29	278	871	399	957	726	726
Sokoban	29	11	18	57	54	18	19
Solitaire	22	2,374	4,357	317	1,610	2,494	631
WeightDomSet	29	8	2,649	1,147	5,441	3,572	248
WireRouting	23	87	535	144	289	1,223	2,103
Total	377	11,934	26,578	10,066	18,965	33,281	12,773

Comparing *claspfolio*, trained on all available instances, with *claspfolio*^{v}, where training and test sets are disjoint, we see that applying *claspfolio*$^{(v)}$ to unseen instances yields lower prediction quality. If the training set represents the dependencies between features and runtime rather loosely, the regression models hardly generalize to unseen instances, which obstructs a good parameter selection. But even in this case, *claspfolio*^{v} is almost twice as fast as *clasp*, which shows that the trained models are still helpful.

In Table 2, we compare *claspfolio* with *paramils*, an automatic configuration tool based on iterated local search (*FocusedILS*) through the configuration space. Given that *paramils* uses a model-free approach, it can only generalize between homogeneous problem classes regarding the best configuration. In contrast, *claspfolio* is utterly applicable to heterogeneous classes in view of its regression models. To reflect this discrepancy, the column *paramils*^{c} shows the runtimes of the best configurations of *clasp* determined by *paramils* independently for each problem class, while the best configuration found over all problem classes is displayed in column *paramils*^{a}. In both cases, we ran four (randomized) copies of *paramils* for 24 hours with a timeout of 600 seconds per run on an instance, as suggested in [6], and then selected the best configuration found. Also note that, in view of only 377 instances evaluated overall, we did not split instances into a training and a test set, i.e., *paramils* was used to automatically analyze *clasp* configurations rather than predicting their performances.

As it could be expected, the configurations found by *paramils*^{c} are much faster than the global one of *paramils*^{a}. On some problem classes, e.g., WeightDomSet, *paramils*^{c} found configurations that rendered the classes almost trivial to solve. On such classes, the configurations of *paramils*^{c} also yield much better performances than the ones of *claspfolio* and *clasp*^{m}. However, on problem classes including very hard instances, like GraphColouring and Solitaire, the configurations determined by *paramils* were less successful. This can be explained by long runs on instances, so that fewer configurations could be explored by local search within the allotted 24 hours.

Comparing *claspfolio* and *paramils*^{c}, *claspfolio* performs better in total, yet worse on ten of the thirteen classes. One reason is that *claspfolio* is based on a small set of configurations, whereas *paramils* considers a much larger configuration space (about 10^{12} configurations). In addition, *paramils*^{c} determined a suitable configuration individually for each class, while *claspfolio* applies the same configurations and models to all problem classes. In fact, we note that *claspfolio*^{v} performs better than *paramils*^{a}. From this, we conclude that the problem classes are heterogeneous, so that it is unlikely to find a single configuration well-suited for all classes. Thus, *claspfolio* appears to be a reasonable approach for configuring *clasp* for sets of heterogeneous instances.

4 Discussion

In this preliminary report, we described a simple yet effective way to counterbalance the sensitivity of ASP solvers to parameter configuration. As a result, ASP solving regains a substantial degree of declarativity insofar as users may concentrate on problem posing rather than parameter tuning. The resulting portfolio-based solver *claspfolio* largely improves on the default configuration of the underlying ASP solver *clasp*. Moreover, our approach outperforms a manual one conducted by experts.

Although our approach is inspired by *satzilla*, *claspfolio* differs in several ways. Apart from the different areas of application, SAT vs. ASP, *satzilla*'s learning and selection engine relies on Ridge Regression, while ours uses Support Vector Regression. Interestingly, *satzilla* incorporates a SAT/UNSAT likelihood prediction further boosting its performance. Our first experiments in this direction did not have a similar effect, and it remains future work to investigate the reasons for this.

Our experiments emphasize that search for an optimal configuration, e.g., via *paramils* using local search, on one (homogeneous) problem class is more effective than *claspfolio*. But the search time of *paramils* for each problem class makes *claspfolio* more efficient on a set of (heterogeneous) problem classes. In fact, predicting a good configuration with *claspfolio* is almost instantaneous, once the regression models are trained. A recent approach to learn domain-specific decision heuristics [10] requires modifying a solver in order to learn and apply the heuristics.

It is interesting future work to investigate automatic portfolio generation. New configurations, to add to a portfolio, could be found with *paramils*. First attempts are done with *hydra* [11]. Further related work includes [12,13,14,15,16], whose discussion is however beyond the scope of this paper. Another goal of future work includes the investigation and selection of the extracted features to predict more precisely the runtime. Usually, feature selection decreases the prediction error of machine learning algorithms. In view of this, the potential of *claspfolio* is not yet fully harnessed in its current version.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2. We are grateful to Holger H. Hoos and Frank Hutter for fruitful discussions on the subject of this paper.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007, pp. 386–392. AAAI Press, Menlo Park (2007)
3. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning SAT solvers with restarts. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 654–668. Springer, Heidelberg (2009)
4. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR 32, 565–606 (2008)
5. Basak, D., Pal, S., Patranabis, D.: Support vector regression. NIP 11(10), 203–224 (2007)
6. Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. JAIR 36, 267–306 (2009)
7. `http://potassco.sourceforge.net`
8. `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
9. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
10. Balduccini, M.: Learning domain-specific heuristics for answer set solvers. In: ICLP 2010 Tech. Comm., pp. 14–23 (2010)
11. Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: Automatically configuring algorithms for portfolio-based selection. In: AAAI 2010, pp. 210–216. AAAI Press, Menlo Park (2010)
12. Gagliolo, M., Schmidhuber, J.: Learning dynamic algorithm portfolios. AMAI 47(3-4), 295–328 (2006)
13. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: AAAI 2007, pp. 255–260. AAAI Press, Menlo Park (2007)
14. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: AICS 2008 (2008)
15. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified Boolean formulas. Constraints 14(1), 80–116 (2009)
16. Arbelaez, A., Hamadi, Y., Sebag, M.: Continuous search in constraint programming. In: ICTAI 2010, pp. 53–60. IEEE Press, Los Alamitos (2010)

plasp: A Prototype for PDDL-Based Planning in ASP

Martin Gebser, Roland Kaminski, Murat Knecht, and Torsten Schaub*

Institut für Informatik, Universität Potsdam

Abstract. We present a prototypical system, *plasp*, implementing Planning by compilation to Answer Set Programming (ASP). Our approach is inspired by Planning as Satisfiability, yet it aims at keeping the actual compilation simple in favor of modeling planning techniques by meta-programming in ASP. This has several advantages. First, ASP modelings are easily modifiable and can be studied in a transparent setting. Second, we can take advantage of available ASP grounders to obtain propositional representations. Third, we can harness ASP solvers providing incremental solving mechanisms. Finally, the ASP community gains access to a wide range of planning problems, and the planning community benefits from the knowledge representation and reasoning capacities of ASP.

1 Introduction

Boolean Satisfiability (SAT; [1]) checking provides a major implementation technique for Automated Planning [2]. In fact, a lot of efforts have been made to develop compilations mapping planning problems to propositional formulas. However, the underlying techniques are usually hard-wired within the compilers, so that further combinations and experiments with different features are hard to implement.

We address this situation and propose a more elaboration-tolerant platform to Planning by using Answer Set Programming (ASP; [3]) rather than SAT as target formalism. The idea is to keep the actual compilation small and model as many techniques as possible in ASP. This approach has several advantages. First, planning techniques modeled in ASP are easily modifiable and can be studied in a transparent setting. Second, we can utilize available ASP grounders to obtain propositional representations. Third, we can harness ASP solvers providing incremental solving mechanisms. Finally, the ASP community gains access to a wide range of planning problems, and the planning community benefits from the knowledge representation and reasoning capacities of ASP.

Our prototypical system, *plasp*, follows the approach of *SATPlan* [4,5] in translating a planning problem from the Planning Domain Definition Language (PDDL; [6]) into Boolean constraints. Unlike *SATPlan*, however, we aim at keeping the actual compilation simple in favor of modeling planning techniques by meta-programming in ASP. Although the compilations and meta-programs made available by *plasp* do not yet match the sophisticated approaches of dedicated planning systems, they allow for applying ASP systems to available planning problems. In particular, we make use of the incremental ASP system *iClingo* [7], supporting the step-wise unrolling of problem horizons. Our case studies demonstrate the impact of alternative compilations and ASP modelings on the performance of *iClingo*.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

2 Architecture

As illustrated in Figure 1, *plasp* translates a PDDL problem instance to ASP and runs it through a solver producing answer sets. The latter represent solutions to the initial planning problem. To this end, a plan is extracted from an answer set and output in PDDL syntax. *plasp* thus consists of two modules, viz., the ASP and Solution compilers. The *ASP compiler* is illustrated in Figure 2. First, a parser reads the PDDL description as input and builds an internal representation, also known as Abstract Syntax Tree (AST). Then, the *Analyzer* gathers information on the particular problem instance; e.g., it determines predicates representing fluents. Afterwards, the *Preprocessor* modifies the instance and enhances it for the translation process. Finally, the *ASP backend* produces an ASP program using the data gathered before. The *Solution compiler* constructs a plan from an answer set output by the solver. This is usually just a syntactic matter, but it becomes more involved in the case of parallel planning where an order among the actions must be re-established. Afterwards, the plan is verified and output in PDDL syntax.

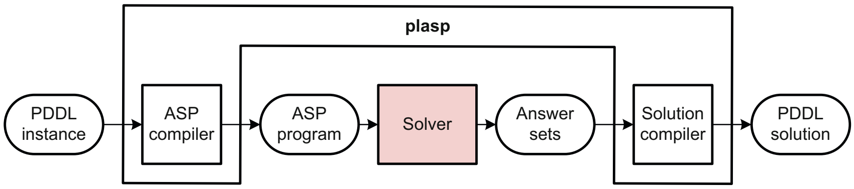


Fig. 1. Architecture of the *plasp* system

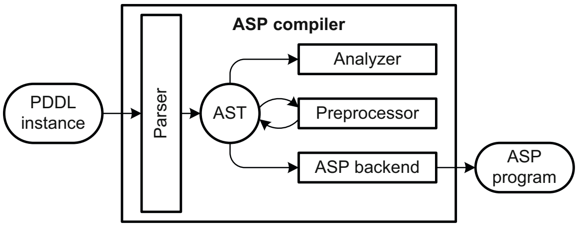


Fig. 2. Architecture of the *ASP compiler*

3 Compilations and Meta-Programs

In order to give an idea of the resulting ASP programs, let us sketch the most basic planning encoding relying on meta-programming. To this end, a PDDL domain description is mapped onto a set of facts built from predicates $init$, $goal$, $action$, $demands$, $adds$, and $deletes$ along with their obvious meanings. Such facts are then combined with the meta-program in Figure 3. Note that this meta-program is treated incrementally by the ASP system *iClingo*, as indicated in lines (1), (3), and (10). While the facts resulting from the initial PDDL description along with the ground rules of (2) are processed just once, the rules in (4)–(9) are successively grounded for increasing values of t and accumulated in *iClingo*'s solving component. Finally, goal conditions are expressed by

$$
\begin{array}{rl}
(1) & \#base. \\
(2) & holds(F,0) \leftarrow init(F). \\
(3) & \#cumulative\ t. \\
(4) & 1\ \{apply(A,t) : action(A)\}\ 1. \\
(5) & \leftarrow apply(A,t), demands(A,F,true), not\ holds(F,t{-}1). \\
(6) & \leftarrow apply(A,t), demands(A,F,false), holds(F,t{-}1). \\
(7) & holds(F,t) \leftarrow apply(A,t), adds(A,F). \\
(8) & del(F,t) \leftarrow apply(A,t), deletes(A,F). \\
(9) & holds(F,t) \leftarrow holds(F,t{-}1), not\ del(F,t). \\
(10) & \#volatile\ t. \\
(11) & \leftarrow goal(F,true), not\ holds(F,t). \\
(12) & \leftarrow goal(F,false), holds(F,t).
\end{array}
$$

Fig. 3. Basic ASP encoding of STRIPS planning

$$
\begin{array}{rl}
(4') & 1\ \{apply(A,t) : action(A)\}. \\
(4'a) & \leftarrow apply(A_1,t), apply(A_2,t), A_1 \neq A_2, demands(A_1,F,true), deletes(A_2,F). \\
(4'b) & \leftarrow apply(A_1,t), apply(A_2,t), A_1 \neq A_2, demands(A_1,F,false), adds(A_2,F). \\
(4'c) & \leftarrow apply(A_1,t), apply(A_2,t), A_1 \neq A_2, adds(A_1,F), deletes(A_2,F).
\end{array}
$$

Fig. 4. Adaptation of the basic ASP encoding to parallel STRIPS planning

volatile rules, contributing ground rules of (11) and (12) only for the current step t. See [7] for further details on incremental ASP solving. From a representational perspective, it is interesting to observe that ASP allows for omitting a frame axiom (like the one in line (9)) for negative information, making use of the fact that instances of *holds* are false by default, that is, unless they are explicitly derived to be true. Otherwise, the specification follows closely the semantics of STRIPS [2].

Beyond the meta-program in Figure 3, *plasp* offers planning with concurrent actions. The corresponding modification of the rule in (4) is shown in Figure 4. While $(4')$ drops the uniqueness condition on applied actions, the additional integrity constraints stipulate that concurrent actions must not undo their preconditions, nor have conflicting effects. The resulting meta-program complies with the $\forall$-step semantics in [8]. Furthermore, *plasp* offers operator splitting as well as forward expansion. The goal of operator splitting [9] is to reduce the number of propositions in the representation of a planning problem by decomposing action predicates; e.g., an action $a(X,Y,Z)$ can be represented in terms of $a_1(X), a_2(Y), a_3(Z)$. Forward expansion (without mutex analysis [10]) instantiates schematic actions by need, viz., if their preconditions have been determined as feasible at a time step, instead of referring to statically given instances of the *action* predicate. This can be useful if initially many instances of a schematic action are inapplicable, yet it requires a domain-specific compilation; meta-programming is difficult to apply because *action* instances are not represented as facts. Finally, *plasp* supports combinations of forward expansion with either concurrent actions or operator splitting. Regardless of whether forward expansion is used, concurrent actions and operator splitting can currently not be combined; generally, both techniques are in opposition, although possible solutions have recently been proposed [11].

4 Experiments

We conducted experiments comparing the different compilation techniques furnished by *plasp*[1] (1.0): the meta-program in Figure 3 (column "basic" in Table 1), its adaptation to concurrent actions in Figure 4 ("concur"), operator splitting ("split"), forward expansion ("expand"), and two combinations thereof ("concur+expand" and "split+expand"). To compute answer sets of compilations, representing shortest plans, *plasp* uses (a modified version of) the incremental ASP system *iClingo*[1] (2.0.5). Although we mainly study the effect of different compilations on the performance of *iClingo*, for comparison, we also include *SATPlan*[2] (2006) and *SGPlan*[3] (5.2.2). While *SGPlan* [12] does not guarantee shortest plan lengths, the approach of *SATPlan*, based on compilation and the use of a SAT solver as search backend, leads to shortest plans. In fact, its compilation is closely related to the "concur+expand" setting of *plasp*, where *SATPlan* in addition applies mutex analysis. The benchmarks, formulated in the STRIPS subset[4] of PDDL, stem from the Second International Planning Competition[4], except for the three Satellite instances taken from the fourth competition[5]. All experiments were run on a Linux PC equipped with 2 GHz CPU and 2 GB RAM, imposing 900 seconds as time and 1.5 GB as memory limit.

Runtime results in seconds are shown in Table 1; an entry "—" indicates a timeout, and "mem" stands for memory exhaustion. On all benchmarks but Schedule, we observe that *SGPlan* has an edge on the other, less specialized (yet guaranteeing shortest plans) systems. The fact that *SATPlan* is usually faster than *plasp* can be explained by the fact that compilations of *plasp* are instantiated by a general-purpose ASP grounder, while *SATPlan* utilizes a planning-specific frontend [10]. Moreover, mutex analysis as in *SATPlan* is currently not included in (encodings of) *plasp*. However, we observe that different compilation techniques of *plasp* pay off on particular benchmarks. On the Blocks and small Elevator instances, the simplest meta-program ("basic") is superior because concurrency and expansion are barely applicable to them and may even deteriorate performance. On Elevator-5-0, splitting ("split") helps to reduce the size of the problem representation. Furthermore, we observe that allowing for concurrent actions without explicit mutexes ("concur" and "concur+expand") dramatically decreases search efficiency on the Elevator domain. However, concurrent actions in combination with forward expansion ("concur+expand") are valuable on FreeCell and Logistics instances, given that they involve non-interfering actions. Splitting ("split" and "split+expand") appears to be useful on Satellite instances, where Satellite-2 again yields the phenomenon of concurrent actions deteriorating search. Finally, forward expansion ("expand") enables *plasp* to successfully deal with the Schedule domain, where even *SATPlan* and *SGPlan* exceed the memory limit. We conjecture that (too) exhaustive preprocessing, e.g., mutex analysis, could be responsible for this.

[1] http://potassco.sourceforge.net
[2] http://www.cs.rochester.edu/~kautz/satplan
[3] http://manip.crhc.uiuc.edu/programs/SGPlan
[4] http://www.cs.toronto.edu/aips2000
[5] http://www.tzi.de/~edelkamp/ipc-4

Table 1. Experiments comparing different compilations

Benchmark	basic	concur	split	expand	concur+expand	split+expand	SATPlan	SGPlan
Blocks-4-0	0.16	0.21	0.43	0.21	0.22	0.20	0.34	0.10
Blocks-6-0	0.30	0.63	0.93	0.44	0.56	1.40	0.27	0.04
Blocks-8-0	1.58	6.53	12.78	98.60	317.53	47.57	1.24	0.09
Elevator-3-0	0.27	0.56	0.92	0.30	0.46	0.89	0.10	0.02
Elevator-4-0	11.72	264.11	20.30	14.69	324.18	28.88	0.30	0.02
Elevator-5-0	—	—	320.58	—	—	467.98	0.61	0.04
FreeCell-2-1	93.42	mem	64.28	60.52	51.33	56.94	2.44	0.12
FreeCell-3-1	—	mem	—	—	175.03	—	10.44	0.14
Logistics-4-0	7.85	0.38	79.15	8.81	0.39	70.56	0.34	0.05
Logistics-7-0	—	0.99	—	—	0.61	—	0.31	0.04
Logistics-9-0	—	0.89	—	—	0.57	—	0.27	0.04
Satellite-1	0.23	0.87	0.23	0.29	0.74	0.26	0.10	0.03
Satellite-2	4.56	638.08	2.19	5.43	448.60	2.69	0.41	0.03
Satellite-3	8.76	3.52	4.00	7.54	3.29	3.70	0.21	0.04
Schedule-2-0	mem	mem	mem	1.03	3.37	mem	mem	mem
Schedule-3-0	mem	mem	mem	1.63	12.89	mem	mem	mem

In summary, we conclude that the different compilation techniques of *plasp* can be advantageous. The automatic, domain-specific choice of an appropriate compilation, required in view of varying characteristics [12], is an intrinsic subject to future work.

5 Discussion

We have presented a prototypical approach to Automated Planning by means of compilation to ASP. In order to close the gap to established planning systems, more background knowledge (e.g., mutexes) would need to be included. If such knowledge can be encoded in meta-programs, it fosters elaboration tolerance and flexibility of planning implementations. In fact, the recent version transition of *iClingo* from 2 to 3 gives inherent support of forward expansion, generating the possibly applicable instances of actions (and fluents) on-the-fly during grounding. Importantly, regardless of additional features that might boost performance (cf. [13]), the compilation capacities of *plasp* are already useful as they make various planning problems, formulated in PDDL, accessible as benchmarks for ASP systems. The range could be further extended by generalizing the compilations supported by *plasp* beyond the STRIPS subset of PDDL.

Given the proximity of Planning and General Game Playing (GGP; [14]), the latter can also (partially) be implemented by compilation to ASP. An approach to solve single-player games in ASP is provided in [15], and [16] presents ASP-based methods to prove properties of games, which can then be exploited for playing. Automatically proving properties of interest to steer the selection of solving techniques may also be useful for Planning. Another line of future work could be Conformant Planning [17], whose elevated complexity could be addressed by compilation to disjunctive ASP. In fact, the *dlv*$^{\mathcal{K}}$ system [18] supports Conformant Planning wrt action language $\mathcal{K}$.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2.

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. IOS Press, Amsterdam (2009)
2. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann, San Francisco (2004)
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
4. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI 1992), pp. 359–363. Wiley, Chichester (1992)
5. Kautz, H., Selman, B.: Pushing the envelope: Planning, propositional logic, and stochastic search. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996), pp. 1194–1201. AAAI/MIT Press (1996)
6. McDermott, D.: PDDL — the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
7. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 190–205. Springer, Heidelberg (2008)
8. Rintanen, J., Heljanko, K., Niemelä, I.: Parallel encodings of classical planning as satisfiability. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 307–319. Springer, Heidelberg (2004)
9. Kautz, H., McAllester, D., Selman, B.: Encoding plans in propositional logic. In: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR 1996), pp. 374–384. Morgan Kaufmann, San Francisco (1996)
10. Blum, A., Furst, M.: Fast planning through planning graph analysis. Artificial Intelligence 90(1-2), 279–298 (1997)
11. Robinson, N., Gretton, C., Pham, D., Sattar, A.: SAT-based parallel planning using a split representation of actions. In: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009), pp. 281–288. AAAI Press, Menlo Park (2009)
12. Hsu, C., Wah, B., Huang, R., Chen, Y.: Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 1924–1929. AAAI/MIT Press (2007)
13. Sideris, A., Dimopoulos, Y.: Constraint propagation in propositional planning. In: Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010), pp. 153–160. AAAI Press, Menlo Park (2010)
14. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
15. Thielscher, M.: Answer set programming for single-player games in general game playing. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 327–341. Springer, Heidelberg (2009)
16. Thielscher, M., Voigt, S.: A temporal proof system for general game playing. In: Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI 2010), pp. 1000–1005. AAAI Press, Menlo Park (2010)
17. Smith, D., Weld, D.: Conformant Graphplan. In: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI 1998), pp. 889–896. AAAI/MIT Press (1998)
18. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning. Artificial Intelligence 144(1-2), 157–211 (2003)

Cluster-Based ASP Solving with *claspar*

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub*, and Bettina Schnor

Institut für Informatik, Universität Potsdam

Abstract. We report on three recent advances in the distributed ASP solver *claspar*. First, we describe its flexible architecture supporting various search strategies, including competitive search using a portfolio of solver configurations. Second, we describe *claspar*'s distributed learning capacities that allow for sharing learned nogoods among solver instances. Finally, we discuss *claspar*'s approach to distributed optimization.

1 Introduction

In view of the rapidly growing availability of clustered, multi-processor, and/or multi-core computing devices, we developed in [1] the distributed ASP solver *claspar*, allowing for the parallelization of the search for answer sets by appeal to the ASP solver *clasp* [2]. *claspar* relies on the Message Passing Interface (MPI; [3]), realizing communication and data exchange between computing units via message passing. Interestingly, MPI abstracts from the actual hardware and lets us execute our system on clusters as well as multi-processor and/or multi-core machines.

This paper reports on the progress made since the first system description of *claspar* [1] covering the features of version 0.1.0: it mainly dealt with the communication in its simple initial master-worker architecture along with a first empirical evaluation of *claspar*'s performance. This early version of *claspar* used the well-known *guiding path* technique [4] for splitting the search space into disjoint parts. Apart from finding a single answer set, *claspar* (0.1.0) also allowed for enumerating answer sets by combining the scheme in [5] with the aforementioned guiding path technique.

2 Advances in *claspar*

We focus in what follows on the major novelties of the current *claspar* version 0.9.0 wrt to the one in [1]. We presuppose some basic knowledge in conflict-driven ASP solving and an acquaintance with concepts like nogoods, decision levels, restarts, etc. The interested reader is referred to [2] for details.

2.1 Search

The simple master-worker architecture of *claspar* (0.1.0) has been extended in order to provide more flexible communication topologies for enhancing *claspar*'s scalability as well as different search strategies. To begin with, a hierarchical master-worker

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

structure can be defined (with option `--topology=<arg>`) that consists of a single superior master along with further inferior masters, each controlling a group of workers. That is, giving argument `master-worker` enforces a flat hierarchy by making one master control all workers, while '`hierarchical,<n>`' creates $\lfloor (p-2)/\mathtt{n} \rfloor$ inferior masters, each controlling `n`$-$`1` workers, where p is the overall number of processes.

claspar provides two major search strategies. The first one aims at partitioning the search space by appeal to the guiding path technique. The second one aims at a competitive search for answer sets. To this end, the aforementioned (flat) topology can be further refined by providing argument '`competition,<n>`' for associating each worker with `n`$-$`1` competitors dealing with the same search space.

Competitive search is supported by the so-called `--portfolio-mode`, making competitors run with different settings. To this end, the option `--portfolio-file` allows for specifying a portfolio of different *clasp* configurations, either a predefined one via `preset` or a handcrafted one read from a given file. These configurations are then attributed to the aforementioned competitors, either randomly or in a `round-robin` fashion, depending on the argument passed to `--portfolio-mode`. Notably, this portfolio mode can be combined with the guiding path method, restricted to the communication between the master and workers.

Otherwise, *claspar* supports all search options of *clasp*, thus making each competitor highly configurable.

2.2 Nogood Exchange

Given that each *clasp* instance relies on *conflict-driven nogood learning* [2], in a distributed setting, it becomes interesting to exchange learned nogoods among different solver instances. This feature adds another degree of freedom to ASP solving and must be handled with care because each instance may learn exponentially many nogoods, so that their distribution may lead to overwhelmingly many nogoods significantly hampering the overall performance.

The exchange of nogoods in *claspar* is configured through two options, viz. `--nogood-sharing` and `--nogood-distribution`. While the former allows for filtering nogoods for exchange, the latter specifies the topology of the exchange.

The export of a nogood is either subject to its number of literals (`length`) or the number of distinct decision levels associated with its literals (`lbd`; cf. [6]). In both cases, smaller values are regarded as advantageous since they are prone to prune larger parts of the search space. Moreover, their exchange can be restricted to packages rather than individual nogoods in order to reduce communication. Finally, *claspar* allows us to restrict the integration to not yet satisfied nogoods. The default configuration of *claspar*'s nogood exchange is '`lbd,3,300,True`', sharing each time 300 nogoods at once, each with at most three different decision levels, no matter whether they are satisfied.

The second option specifies the topology of nogood exchange; it distinguishes four different settings:

`none` disables nogood exchange;

`local` enables nogood exchange between a worker and its competitors, or workers sharing an inferior master (provided that the corresponding search topology is set). Otherwise, this option is equivalent to exchange among all solver instances;

`cube` organizes all solver instances in a hypercube and allows for nogood exchange between connected instances. This topology is particularly suited for large numbers of workers because each node in a hypercube has at most logarithmically many neighbors;

`all` engages nogood exchange among all solver instances.

2.3 Optimization

Apart from the basic reasoning modes of finding and enumerating answer sets, *claspar* now also supports optimization. To this end, it allows for exchanging upper bounds of objective functions, similar to the exchange of nogoods. In more detail, this works as follows. Whenever a *clasp* instance finds an answer set, it sends it along with its objective value(s) to a printer process. In turn, the printer process writes the answer set to the console and broadcasts the current upper bound to all *clasp* instances, which then integrate a corresponding constraint. If a local upper bound is larger, the solver instance engages a restart and updates its bound; otherwise, the running search is continued.

3 Experiments

Our experiments were run on a cluster of 28 nodes, each equipped with two quad-core Intel Xeon E5520 processors and 48GB main memory. In view of this, we ran 8 processes per node, where nodes are connected via InfiniBand (20Gb/s). We evaluated *claspar* version 0.9.0, having two distinguished processes, viz. a *master* and a *printer*. The benchmark set consists of 68 ASP (ca. 90% unsatisfiable) and 78 SAT (ca. 60% unsatisfiable) problems, mostly taken from the last two respective competitions and running at least 100s with *clasp* on a standard PC. Each entry in the below tables reflects the sum of runtimes (wall clock) per problem category (and numbers of timed-out runs in parentheses), where timeouts are taken at and counted as 4000s in Section 3.1 and 3.2 or 600s in Section 3.3, respectively. Detailed results are available at [7].

3.1 Search

For evaluating the different search strategies of *claspar*, we considered three different configurations. Their common base consists of workers amounting to *clasp* version 1.3.6 plus the two aforementioned special-purpose processes. Numbers of workers and nodes are given in the heading of Table 1, where 'w+2 (n)' indicates that we ran w solving processes and two controlling processes over n nodes. The configurations include:

Guiding path applies the guiding path strategy to all available workers running with *clasp*'s default settings. Hence, w disjoint search spaces are addressed in parallel.

Uniform portfolio combines guiding path with competitive search in having groups of up to 8 workers under the same guiding path. Accordingly, n disjoint search spaces are addressed in parallel. The competing solvers run *clasp*'s default settings with different random seeds in order to increase their variation (already inherent due to race conditions).

Table 1. Comparison of different search strategies

		1+2 (1)	6+2 (1)	30+2 (4)	62+2 (8)	126+2 (16)
Guiding path	ASP	174,661 (24)	154,504 (22)	103,283 (14)	85,578 (11)	71,799 (8)
	SAT	89,428 (8)	42,491 (5)	38,293 (6)	30,515 (4)	28,916 (5)
	all	264,090 (32)	196,995 (27)	141,577 (20)	116,094 (15)	100,715 (13)
Uniform portfolio	ASP	174,661 (24)	149,157 (17)	133,147 (18)	113,309 (16)	96,466 (13)
	SAT	89,428 (8)	57,694 (3)	40,555 (2)	31,734 (2)	26,020 (2)
	all	264,090 (32)	206,851 (20)	173,702 (20)	145,043 (18)	122,486 (15)
Non-uniform portfolio	ASP	174,661 (24)	141,890 (16)	98,160 (11)	92,331 (11)	71,709 (8)
	SAT	89,428 (8)	52,739 (3)	37,772 (3)	30,739 (1)	22,528 (1)
	all	264,090 (32)	194,629 (19)	135,932 (14)	123,071 (12)	94,237 (9)

Non-uniform portfolio is identical to the previous configuration except that it uses a handcrafted portfolio for competitive search. The portfolio consists of the following *clasp* settings, chosen to cover diverse search strategies and heuristics:

- *default*
- *default* + `--berk-max=512 --berk-huang=yes`
- *default* + `--save-progress=1`
- *default* + `--restarts=128 --local-restart=1`
- *default* + `--restarts=128 --save-progress=1`
- *default* + `--restarts=256`
- *default* + `--restarts=256 --save-progress=1`
- *default* + `--heuristic=VSIDS`

Looking at Table 1, we observe a different comportment on benchmarks stemming from SAT and ASP. While **non-uniform portfolio** solving seems to have a clear edge on SAT problems, it behaves equally well as the **guiding path** strategy on ASP benchmarks. This may be due to the fact that ASP problems tend to have higher combinatorics than SAT problems, so that they are better suited for being split into several subproblems. Although we generally observe performance improvements with increasing number of workers, the speed-ups are not (near to) linear. Linear speed-ups were still obtained with the **guiding path** strategy applied to particular problems, such as pigeon-hole instances included in the ASP category. A detailed investigation of further benchmarks sometimes yields super-linear speed-ups, even on unsatisfiable problems, as well as slow-downs. In fact, the latter hint at a lack of learned nogood exchange, which is considered next.

3.2 Nogood Exchange

For simplicity, we investigate nogood exchange on top of the most successful strategy of the previous section, viz. **non-uniform portfolio** search. Of the four options from Section 2.2, we dropped nogood exchange among `all` solver instances because our preliminary experiments showed that this option is not competitive for larger numbers of workers. The results for the `none` option are identical to those in Table 1. Option `local` restricts nogood exchange to workers addressing the same search space,

Table 2. Comparison of different nogood exchange strategies

		1+2 (1)	6+2 (1)	30+2 (4)	62+2 (8)
none	ASP	174,661 (24)	141,890 (16)	98,160 (11)	92,331 (11)
	SAT	89,428 (8)	52,739 (3)	37,772 (3)	30,739 (1)
	all	264,090 (32)	194,629 (19)	135,932 (14)	123,071 (12)
local	ASP	174,661 (24)	93,166 (11)	75,678 (13)	58,747 (7)
	SAT	89,428 (8)	29,067 (0)	28,324 (3)	14,373 (1)
	all	264,090 (32)	122,234 (11)	104,002 (16)	73,120 (8)
cube	ASP	174,661 (24)	92,108 (10)	82,388 (13)	64,028 (9)
	SAT	89,428 (8)	27,245 (0)	33,602 (4)	24,099 (2)
	all	264,090 (32)	119,354 (10)	115,991 (17)	88,128 (11)

whereas `cube` allows for more global exchange by connecting workers beyond groupings. In Table 2, we observe that nogood exchange clearly improves over `none`, especially for the column headed by ‘6+2 (1)’, refraining from search space splitting. This is particularly interesting for desktop machines offering only limited multi-processing capacities. Almost no further improvements are observed by quadrupling the number of nodes, with workers under four distinct guiding paths. In order to achieve further speed-ups due to search space splitting, we had to increase the number of workers significantly, as shown in the last column. Here, the `local` exchange has an edge on the more global `cube`-oriented exchange. This suggests that sharing among solvers treating the same subproblem promotes the relevance of the exchanged nogoods.

3.3 Optimization

To evaluate the optimization capacities of *claspar*, we consider a collection of 53 hard problems from the last ASP competition [8], each involving exactly one optimization criterion. Given that most of the runs timed out after 600s, we rank each configuration by the score $[(\text{shortest runtime}/\text{runtime}) * (\text{lowest upper bound}/\text{upper bound})]$ per problem (zero if no answer set found at all). Note that greater scores are better than smaller ones, and the sums of scores (and numbers of timed-out runs in parentheses) are provided in Table 3.

For the considered benchmark collection, the pure **guiding path** strategy performed best overall and exhibited the smallest number of timeouts with 62 workers. In fact, the benchmarks include one problem class (15PuzzleOpt) such that, for many of its instances, solutions could in time be proven to be optimal only with the **guiding path** strategy. Note that the **guiding path** configurations rely on *clasp*’s default settings, including a rather slow restart policy. On the other hand, the **non-uniform portfolio** approach involves rapid restart policies that are not very helpful here because the investigated optimization problems are highly combinatorial. Interestingly, the **uniform portfolio** strategy nonetheless failed to achieve significant improvements.

Finally, we ran the pure **guiding path** strategy with `cube`-oriented nogood exchange. Unfortunately, the exchange led to performance degradation, which could be related to the fact that the decision variants of the majority of the considered optimization problems are rather under-constrained. Hence, the nogoods learned by individual

Table 3. Comparison of different optimization strategies

	1+2 (1)	6+2 (1)	30+2 (4)	62+2 (8)
Guiding path	28.68 (39)	36.90 (37)	39.65 (37)	46.42 (32)
Uniform portfolio	28.68 (39)	31.80 (39)	36.71 (37)	39.21 (37)
Non-uniform portfolio	28.68 (39)	32.79 (39)	40.29 (37)	39.91 (37)
Guiding path + `cube`	28.68 (39)	36.04 (37)	37.95 (37)	43.81 (34)

solver instances tend to rule out suboptimal solutions, yet without including much communicable information.

4 Discussion

Although distributed parallel ASP solving has the prospect of gaining significant speedups, it also adds further degrees of freedom that must be handled with care. For one, the physical cluster architecture ought to be taken into account for choosing a search topology. Furthermore, nogood exchange is often valuable, but it may also incur the communication of "gibberish" retarding search. In particular, this applies to combinatorial optimization problems, where the parallel computing power could be utilized most effectively by pure search space splitting without exchange. However, the fine-tuning of *claspar* (0.9.0) is still at an early stage, and further investigations are needed to make better use of the increased flexibility.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2.

References

1. Ellguth, E., Gebser, M., Gusowski, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneidenbach, L., Schnor, B.: A simple distributed conflict-driven answer set solver. In: [9], pp. 490–495
2. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007, pp. 386–392. AAAI Press/The MIT Press (2007)
3. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. The MIT Press, Cambridge (1999)
4. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation 21(4), 543–560 (1996)
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 136–148. Springer, Heidelberg (2007)
6. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI 2009, pp. 399–404. AAAI Press/The MIT Press (2009)
7. `http://www.cs.uni-potsdam.de/claspar`
8. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: [9], pp. 637–654
9. Erdem, E., Lin, F., Schaub, T. (eds.): LPNMR 2009. LNCS, vol. 5753. Springer, Heidelberg (2009)

STeLP – A Tool for Temporal Answer Set Programming[*]

Pedro Cabalar and Martín Diéguez

Department of Computer Science,
University of Corunna (Spain)
{cabalar,martin.dieguez}@udc.es

Abstract. In this paper we present STeLP, a solver for Answer Set Programming with temporal operators. Taking as an input a particular kind of logic program with modal operators (called Splitable Temporal Logic Program), STeLP obtains its set of temporal equilibrium models (a generalisation of stable models for this extended syntax). The obtained set of models is represented in terms of a deterministic Büchi automaton capturing the complete program behaviour. In small examples, this automaton can be graphically displayed in a direct and readable way. The input language provides a set of constructs which allow a simple definition of temporal logic programs, including a special syntax for action domains that can be exploited to simplify the graphical output. STeLP combines the use of a standard ASP solver with a linear temporal logic model checker in order to find all models of the input theory.

1 Introduction

The use of Answer Set Programming (ASP) tools to represent temporal scenarios for Non-Monotonic Reasoning (NMR) has some important limitations. ASP solvers are commonly focused on finite domains and so, representation of time usually involves a finite bound. Typically, variables ranging over transitions or time instants are grounded for a sequence of numbers $0, 1, \ldots, n$ where n is a finite length we must fix beforehand. As a result, for instance, we cannot check the non-existence of a plan for a given planning problem, or that some transition system satisfies a given property for any of its possible executions, or that two representations of the same dynamic scenario are *strongly equivalent* (that is, they always have the same behaviour for any considered narrative length) to mention three relevant examples.

To overcome these limitations, [1] introduced an extension of ASP called *Temporal Equilibrium Logic*. This formalism combines Equilibrium Logic [2] (a logical characterisation of ASP) with Linear Temporal Logic (LTL) [3] and provides a definition of the *temporal equilibrium models* (analogous to stable models) for any arbitrary temporal theory.

[*] This research was partially supported by Spanish MEC project TIN2009-14562-C05-04 and Xunta de Galicia project INCITE08-PXIB105159PR.

In this paper we introduce STeLP[1], a system for computing the temporal equilibrium models of a particular class of temporal theories called *Splitable Temporal Logic Programs* (STLP). This class suffices to cover most frequent examples in ASP for dynamic domains. More importantly, the temporal equilibrium models of an STLP have been shown to be computable in terms of a regular LTL theory (see the companion paper [4]). This feature is exploited by STeLP to call an LTL model checker as a backend.

2 Splitable Temporal Logic Programs

Temporal Equilibrium Logic (TEL) shares the syntax of propositional LTL, that is, propositional formulas plus the unary temporal operators[2] $\Box$ (read as "always"), $\Diamond$ ("eventually") and $\bigcirc$ ("next"). TEL is defined in two steps: first, we define the monotonic logic of *Temporal Here-and-There* (THT); and second, we select TEL models as some kind of minimal THT-models obtaining non-monotonicity. For further details, see [5].

Definition 1 (STLP). *An* initial rule *is an expression of one of the forms:*

$$A_1 \wedge \cdots \wedge A_n \wedge \neg A_{n+1} \wedge \cdots \wedge \neg A_m \rightarrow A_{m+1} \vee \cdots \vee A_s \tag{1}$$

$$B_1 \wedge \cdots \wedge B_n \wedge \neg B_{n+1} \wedge \cdots \wedge \neg B_m \rightarrow \bigcirc A_{m+1} \vee \cdots \vee \bigcirc A_s \tag{2}$$

where A_i are atoms and each B_j can be an atom p or the formula $\bigcirc p$. A dynamic rule *has the form $\Box r$ where r is an initial rule. A* Splitable Temporal Logic Program *(STLP) Π is a set of (initial and dynamic) rules.* ⊠

The *body* (resp. *head*) of a rule is the antecedent (resp. consequent) of its implication connective. As usual, a rule with empty body (that is, $\top$) is called a *fact*, whereas a rule with empty head (that is, $\bot$) is called a called a *constraint*. An initial fact $\top \rightarrow \alpha$ is just written as α. The following theory Π_1 is an STLP:

$$\neg a \wedge \bigcirc b \rightarrow \bigcirc a \qquad \Box(a \rightarrow b) \qquad \Box(\neg b \rightarrow \bigcirc a)$$

In [4] it is shown how the temporal equilibrium models of an STLP Π correspond to the LTL models of $\Pi \cup LF(\Pi)$ where $LF(\Pi)$ are loop formulas adapted from the result in [6]. For further details and a precise definition, see [4].

3 The Input Language

The input programs of STeLP adopt the standard ASP notation for conjunction, negation and implication, so that, an initial rule like (1) is represented as:

$$A_{m+1} \texttt{ v} \ldots \texttt{v } A_s \texttt{ :- } A_1\texttt{,} \ldots \texttt{,} A_n\texttt{, not } A_{n+1}\texttt{,} \ldots \texttt{, not } A_m$$

Operator '$\bigcirc$' is represented as '`o`' whereas a dynamic rule like $\Box(\alpha \rightarrow \beta)$ is written as β `::-` α. Using this notation, program Π_1 becomes:

[1] A STeLP web version is available at http://kr.irlab.org/stelp

[2] As shown in [5], the LTL binary operators $\mathcal{U}$ ("until") and $\mathcal{R}$ ("release") can be removed by introducing auxiliary atoms.

```
o a :- not a, o b.        b ::- a.        o a ::- not b.
```

Constraints in `STeLP` are more general than in STLP: their body can include any arbitrary combination of propositional connectives with `o`, `always` (standing for $\Box$) and `until` (standing for $\mathcal{U}$). The empty head $\bot$ is not represented. For instance, $\Box(\bigcirc a \wedge \neg b \rightarrow \bot)$ and $(\Box\neg g) \rightarrow \bot$ are constraints written as:

```
::- o a, not b.        :- always not g.
```

In `STeLP` we can also use rules where atoms have variable arguments like `p(`X_1,...,X_n`)` and, as happens with most ASP solvers, these are understood as abbreviations of all their ground instances. A kind of *safety* condition is defined for variables occurring in a rule. We will previously distinguish a family of predicates, called *static*, that satisfy the property $\Box(\ p(\overline{X}) \leftrightarrow \bigcirc p(\overline{X})\ )$ for any tuple of elements $\overline{X}$. These predicates are declared using a list of pairs *name/arity* preceded by the keyword `static`. All built-in relational operators `=`, `!=`, `<`, `>`, `<=`, `>=` are implicitly defined as static, having their usual meaning. An initial or dynamic rule is *safe* when:

1. Any variable X occurring in a rule $B \rightarrow H$ or $\Box(B \rightarrow H)$ occurs in some positive literal in B for some static predicate p.
2. Initial rules of the form $B \rightarrow H$ where at least one static predicate occurs in the head H only contain static predicates (these are called *static rules*).

Since static predicates must occur in any rule, `STeLP` allows defining global variable names with a fixed domain, in a similar way to the `lparse`[3] directive `#domain`. For instance, the declaration `domain switch(X).` means that any rule referring to variable `X` is implicitly extended by including an atom `switch(X)` in its body. All predicates used in a domain declaration must be static – as a result, they will be implicitly declared as static, if not done elsewhere.

As an example, consider the classical puzzle where we have a wolf `w`, a sheep `s` and a cabbage `c` at one bank of a river. We have to cross the river carrying at most one object at a time. The wolf eats the sheep, and the sheep eats the cabbage, if no people around. Action `m(X)` means that we move some item `w,s,c` from one bank to the other. We assume that the boat is always switching its bank from one state to the other, so when no action is executed, this means we moved the boat without carrying anything. We will use a unique fluent `at(Y,B)` meaning that `Y` is at bank `B` being `Y` an item or the boat `b`. The complete encoding is shown in Figure 1.

A feature that causes a difficult reading of the obtained automaton for a given STLP is that all the information is represented by formulas that occur as transition labels, whereas states are just given a meaningless name. As opposed to this, in an actions scenario, one would expect that states displayed the fluents information and transitions only contained the actions execution. To make the automaton closer to this more natural representation, we can distinguish predicates representing actions and fluents. For instance, in the previous example,

[3] `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`

```
% Domain predicates
domain item(X), object(Y).
static opp/2.     fluent at/2.      action m/1.
opp(l,r). opp(r,l).     item(w). item(s). item(c).
object(Z) :- item(Z).   object(b).
o at(X,A) ::- at(X,B), m(X), opp(A,B).       % Effect axiom for moving
o at(b,A) ::- at(b,B), opp(A,B).             % The boat is always moving
::- m(X), at(b,A), at(X,B), opp(A,B).        % Action executability
::- at(Y,A), at(Y,B), opp(A,B).              % Unique value constraint
o at(Y,A) ::- at(Y,A), not o at(Y,B),opp(A,B).% Inertia
::- at(w,A), at(s,A), at(b,B), opp(A,B).     % Wolf eats sheep
::- at(s,A), at(c,A), at(b,B), opp(A,B).     % Sheep eats cabbage
a(X) ::- not m(X).                           % Choice rules for action
m(X) ::- not a(X).                           %    execution
::- m(X), item(Z), m(Z), X != Z.             % Non-concurrent actions
at(Y,l).                                     % Initial state
g ::- at(w,r), at(s,r), at(c,r).             % Goal predicate
:- always not g.                             % Goal must be satisfied
```

Fig. 1. Wolf-sheep-cabbage puzzle in STeLP

we would further declare: `action m/1. fluent at/2.` STeLP uses this information so that when all the outgoing transitions from a given state share the same information for fluents, this is information is shown altogether inside the state, and removed from the arc labels. Besides, any symbol that is not an action or a fluent is not displayed (they are considered as auxiliary). As a result of these simplifications, we may obtain several transitions with the same label: if so, they are collapsed into a single one.

4 Implementation

STeLP is a Prolog application that interacts with the standard ASP solver DLV[4] and the LTL model checker SPOT[5]. As shown in Figure 2, it is structured in several modules we describe next.

In a first step, STeLP parses the input program, detecting static and domain predicates, checking safety of all rules, and separating the static rules. It also appends static predicates to rule bodies for all variables with global domain. The set of static rules is fed to DLV to generate some model (among possible) that will provide the extension for all static predicates. The number of static models that STeLP will consider is given as a command line argument. Each static model will generate a different ground program and a different automaton. Once a static model is fixed, STeLP grounds the non-static rules input program. Each ground instance is evaluated and, if the body of the ground rule becomes false, the rule is deleted. Otherwise all static predicates of the rule are deleted from its body.

[4] http://www.dbai.tuwien.ac.at/proj/dlv/

[5] http://spot.lip6.fr/

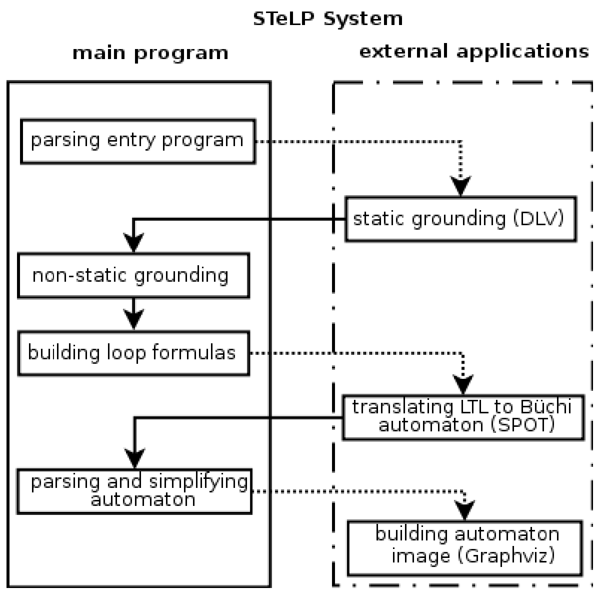


Fig. 2. Structure of `STeLP` system

The next step computes the loop formulas for the ground STLP we have obtained by constructing a dependency graph and obtaining its strongly connected components (the loops) using Tarjan's algorithm [7]. The STLP plus its loop formulas are then used as input for the LTL solver `SPOT`, which returns a deterministic Büchi automaton. Finally, `STeLP` parses and, if actions and fluents are defined, simplifies the automaton as described before. The tool `Graphviz`[6] is used for generating a graphical representation. For instance, our wolf-sheep-cabbage example throws the diagram in Figure 3. As an example of non-existence of plan, if we further include the rule `::- at(w,r), at(c,r), at(b,l)` meaning that we cannot leave the wolf and the cabbage alone in the right bank, then the problem becomes unsolvable (we get a Büchi automaton with no accepting path).

`STeLP` is a first prototype without efficiency optimisations – exhaustive benchmarking is left for future work. Still, to have an informal idea of its current performance, the example above was solved in 0.208 seconds[7]. The automaton for the whole behaviour of the classical scenario involving 3 missionaries and 3 cannibals is solved in 160.358 seconds. `STeLP` is able to show that this same scenario has no solution for 4 individuals in each group, but the answer is obtained in more than 1 hour.

[6] `http://www.graphviz.org/`

[7] Using an Intel Xeon 2.4 GHz, 16 GB RAM and 12 MB of cache size, with software tools SPOT 0.7.1, DLV oct-11-2007 and SWI Prolog 5.8.0.

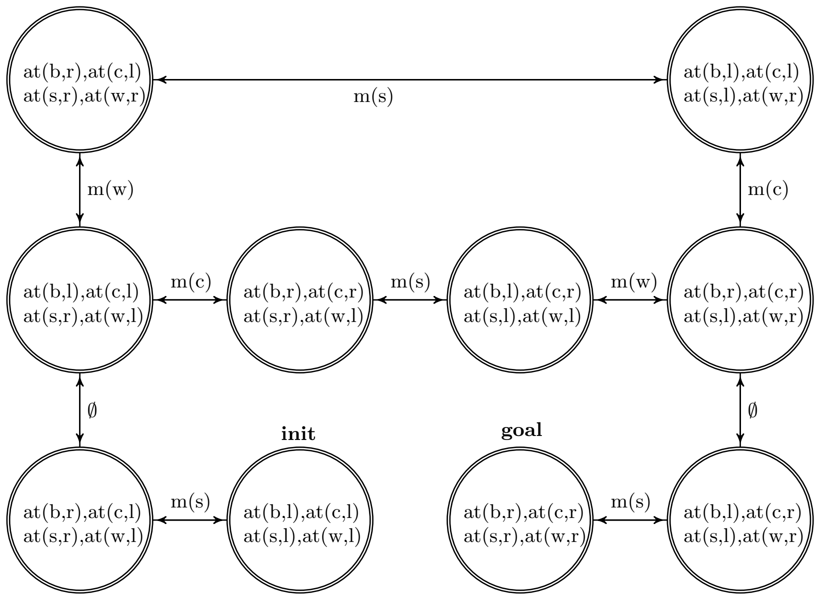


Fig. 3. Automaton for the wolf-sheep-cabbage example

References

1. Cabalar, P., Vega, G.P.: Temporal equilibrium logic: a first approach. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 241–248. Springer, Heidelberg (2007)
2. Pearce, D.: A new logical characterisation of stable models and answer sets. In: Dix, J., Przymusinski, T.C., Moniz Pereira, L. (eds.) NMELP 1996. LNCS, vol. 1216. Springer, Heidelberg (1997)
3. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Heidelberg (1991)
4. Aguado, F., Cabalar, P., Pérez, G., Vidal, C.: Loop formulas for splitable temporal logic programs. In: Delgrande, J., Faber, W. (eds.) LPNMR 2011. LNCS (LNAI), vol. 6645, pp. 78–90. Springer, Heidelberg (2011), `http://www.dc.fi.udc.es/~cabalar/lfstlp.pdf`
5. Cabalar, P.: A normal form for linear temporal equilibrium logic. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 64–76. Springer, Heidelberg (2010)
6. Ferraris, P., Lee, J., Lifschitz, V.: A generalization of the Lin-Zhao theorem. Annals of Mathematics and Artificial Intelligence 47, 79–101 (2006)
7. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)

Compiling Answer Set Programs into Event-Driven Action Rules

Neng-Fa Zhou[1], Yi-Dong Shen[2], and Jia-Huai You[3]

[1] CUNY Brooklyn College & Graduate Center
[2] Institute of Software, Chinese Academy of Sciences
[3] Department of Computing Science, University of Alberta

Abstract. This paper presents a compilation scheme, called ASP2AR, for translating ASP into event-driven action rules. For an ASP program, the generated program maintains a partial answer set as a pair of sets of tuples (called *IN* and *OUT*) and propagates updates to these sets using action rules. To facilitate propagation, we encode each set as a finite-domain variable and treat additions of tuples into a set as events handled by action rules. Like GASP and ASPeRiX, ASP2AR requires no prior grounding of programs. The preliminary experimental results show that ASP2AR is an order of magnitude faster than GASP and is much faster than Clasp on benchmarks that require heavy grounding.

1 Introduction

Most ASP systems such as SModels, ASSAT, CModels, ASPPS, DLV, and Clasp rely on a grounder to transform a given program into a propositional one before computing consistent models called answer sets. This two-phase computation has been considered problematic because the grounding process may take an exponential time (in the size of the non-ground program) and the resulting grounded program may be too large to be stored and processed effectively. The NPDatalog system [2] does not ground programs as these systems, but it translates a source program into a constraint program in OPL which performs complete grounding using iterators.

Recently, a bottom-up iterative approach has been proposed for ASP in which grounding takes place in the computation process [4]. The main idea of this approach is to iteratively apply rule propagation and nondeterministic rule selection until an answer set is found. Two systems, namely, GASP [5] and ASPeRiX [3], have been developed based on this approach. Both systems are interpreters. While these systems outperform the cutting-edge ASP systems such as Clasp on some benchmarks for which grounding is very expensive, they are not as competitive in general.

This paper describes a compiler, called ASP2AR, for translating ASP into event-driven action rules (AR) [6]. For an ASP program, predicates are divided into stratified and unstratified parts. The stratified part is evaluated using tabling and those in the unstratified part are translated into AR to maintain the current interpretation. Like in GASP and ASPeRiX, two disjoint tuple sets, called IN and OUT, are used to represent the current interpretation. To facilitate propagation, we represent the IN and OUT sets for each predicate as two finite-domain variables. In this way, additions of tuples into IN and OUT sets can be treated as events and handled by action rules. When a fixpoint is reached after propagation, a tuple of a negative literal that is neither in IN or OUT is selected and assumed to be true or false. This step is called *labeling*. Labeling a tuple triggers further propagation. This step is repeated until no further tuple can be selected.

Our approach differs from the bottom-up iterative approach used in GASP and ASPeRiX in that it labels negative literals, not rule instances. The answer set semantics requires that every tuple in an answer set must be justified. When a tuple is labeled false, we can safely add it into OUT. When a tuple is labeled true, however, we cannot just add it into IN because it may not be producible by any rule instance. For this reason, we use action rules to ensure that the tuple is supported by at least one rule instance.

Our system is compiler-based. It does not have the interpretation overhead as seen in GASP and ASPeRiX. Compared with propositional programs generated by a grounder, programs generated by our compiler are very small. The preliminary experimental results show that ASP2AR is an order of magnitude faster than GASP and is much faster than Clasp on benchmarks that require heavy grounding; and on the benchmarks that can be grounded easily, however, ASP2AR is still not as competitive as Clasp.

2 The Procedure for Computing Answer Sets

Given a program, we assume that all the predicates that are not dependent on unstratified negation have been completely evaluated using tabling. Figure 1 presents the procedure for computing answer sets when the program contains unstratified negation. Three sets named IN, OUT, and PIN are used to maintain the current partial answer set: IN contains tuples that are known to be true, OUT contains tuples that are known to be false, and PIN contains tuples that have been assumed to be true. The pair $\langle IN, OUT\rangle$ is said to be *inconsistent* if there exists a tuple that is contained in both IN and OUT, and *complete* if for any tuple it is included in either IN or OUT. A tuple of a positive literal is said to be true if it is included in IN, a tuple of a negative literal is said to be true if it is included in OUT, and a tuple is said to be *unknown* if it is included in neither IN nor OUT. In the beginning, OUT and PIN are empty and IN contains all the tuples obtained from the evaluation of the stratified part. The procedure repeatedly applies propagation and labeling until an answer set is found or a failure is reported.

```
compute(){
    initialize $IN$, $OUT$, and $PIN$;
    propagate();
    while ($\langle IN, OUT \rangle$ is consistent but not complete) {
        choose an unknown tuple $t$ of a negative literal;
        labeling: $OUT = OUT \cup \{t\} \oplus PIN = PIN \cup \{t\}$;
        propagate();
    };
    if ($\langle IN, OUT \rangle$ is consistent and every tuple in $PIN$ is also in $IN$)
        output $IN$ as an answer set;
}
propagate(){
    do {
        for (each rule H:-B in the program) {
        right-to-left:
            For each instance of B that is true, add H into $IN$ ;
        left-to-right:
            For each instance of H in $OUT$, ensure no instance of B is true;
        }
    } while (IN or $OUT$ is updated);
    seek-support:
        For each tuple in $PIN$ but not $IN$, ensure it is producible; }
```

Fig. 1. The procedure for computing answer sets

2.1 Right-to-Left Propagation

For each rule "$H{:}{-}B$", the right-to-left propagation adds H into IN for each instance of B that is true. Since all rules are range restricted, H is guaranteed to be ground for each ground instance of B. Let $B =$ "$B_1, \ldots, B_n$". The conjunction of literals is true if all the literals B_i(i=1,...,n) are true. Recall that a ground instance of a positive literals is true if it is included in IN, and a ground instance of a negative literals is true if it is included in OUT. The right-to-left propagation essentially conducts joins of the known relations of the literals B_1, ..., B_n.

2.2 Left-to-Right Propagation

For each rule "$H{:}{-}B$", once a ground instance of H is added into OUT, the left-to-right propagation ensures that no instance of B is true. Let $B =$"$B_1, \ldots, B_n$". If instances of any $n-1$ literals are found to be true, then the instance of the remaining one literal must be false. If the remaining literal is positive, then all of its ground instances are added into OUT and further propagation will ensure that they will never be produced. If the remaining literal is negative, then all of its ground instances are added into PIN, and further propagation will ensure that for each such an instance there exists at least one rule instance that can produce it. Note that a tuple added into PIN is still unknown until it is added into IN afterwards.

2.3 Seek-Support

For each tuple in PIN but not in IN, the seek-support propagation ensures that it has at least one support, i.e., a rule instance, that can produce it. An instance of a positive literal is supported if (1) it is in IN; (2) it is in PIN and has a support; or (3) it is not in PIN and it is unknown. An instance of a negative literal is supported if (1) it is in OUT; or (2) it is unknown. A rule instance "H:$-B_1, \ldots, B_n$" is a support of H if every B_i (i=1,...,n) is supported.

2.4 Labeling

In labeling, an unknown tuple t of a negative literal is chosen and the labeling step nondeterministically adds t into OUT or PIN. When a tuple is labeled false, it is safely added into OUT. After this, propagation will ensure that the tuple can never be produced (*left-to-right*). When a tuple is labeled true, however, it is added into PIN, not IN. After this, propagation will ensure that there exists at least one rule instance that can produce it (*seek-support*).

3 Translation from ASP into AR

The AR (*Action Rules*) language, which was initially designed for programming constraint propagators [6], is used as the target language for compiling ASP. An action rule takes the following form: "$H, G, \{E\}{=}{>}B$" where H (called the *head*) is an atomic formula that represents a pattern for agents, G (called the *guard*) is a conjunction of conditions on the agents, E (called *event patterns*) is a non-empty disjunction of patterns for events that can activate the agents, and B (called *action*) is a sequence of arbitrary subgoals. In general, a predicate can be defined with multiple action rules.

Consider the following example:

```
p(X),{dom_any(X,E)} => writeln(E).
go :- X in 1..4, p(X), X #\= 1, X #\= 3.
```

The event `dom_any(X,E)` is posted whenever an element `E` is excluded from the domain of `X`. The query `go` outputs two lines, 1 in the first line and 3 in the second line.

The stratified part of a given program is completely evaluated using tabling. After this, each remaining predicate must have a known finite domain. We encode each tuple of a predicate as an unique integer and use an integer domain variable to represent each of the IN and OUT sets for the predicate. Initially, the domain contains all the encoded integers for the tuples to denote the empty set. Each time a tuple is added into a set, the encoded integer of the tuple is excluded from the domain. Dummies are added into the domains so that no domain variable will be instantiated. This representation is compact since each element in a domain is encoded as a bit. Furthermore, exclusions of domain elements are treated as `dom_any` events that can be handled using action rules. For the sake of simplicity of presentation, we assume the existence of the event `tuple_added(`$S, p(A_1, ..., A_n)$`)`, which is posted whenever a tuple $p(A_1, ..., A_n)$ is added into the set S.

Consider the ASP rule "`p(X,Z) :-q(X,Y),r(Y,Z).`" To do *right-to-left* propagation, we create an agent named `agent_q` to watch additions of tuples into `INq`. Once a tuple `q(X,Y)` has been added into `INq`, we conduct join of `q(X,Y)` with each of the tuples that have been already added into `INr`. Also, we create another agent named `agent_q_r` to watch future additions of tuples into `INr`. The following defines `agent_q` and `agent_q_r`. The predicate `add_tuple(INp,p(X1,...,Xn))` adds the tuple `p(X1,...,Xn)` into the set `INp`.

```
agent_q(INp,INq,INr),{tuple_added(INq,q(X,Y))} =>
   foreach(r(Y,Z) in INr, add_tuple(INp,p(X,Z))),
   agent_q_r(INp,X,Y,INr).

agent_q_r(INp,X,Y,INr),{tuple_added(INr,r(Y,Z))} =>
   add_tuple(INp,p(X,Z)).
```

Negative literals are treated in the same way as positive literals except that OUT sets are used instead of IN sets.

The compiled program is improved by indexing agents. For example, for the conjunction "`q(X,Y),r(Y,Z)`", after a tuple `q(X,Y)` has been observed, an agent is created to watch only upcoming tuples of `r/2` whose first argument is the same as `Y`. By indexing agents, redundant join operations are avoided.

The *left-to-right* propagation rule is encoded similarly. Consider again the above ASP rule. To ensure that `p(X,Z)` is false, an agent is created to watch `q(X,Y)`. For each such a tuple, `r(Y,Z)` is added into `OUTr`. Another agent is created to watch `r(Y,Z)`. For each such a tuple, `q(X,Y)` is added into `OUTq`.

The *seek-support* rule is encoded in the following way. To ensure that `p(X,Z)` is producible, an agent is created to watch additions of tuples into `OUTq` and `OUTr`. Each time a tuple is added into `OUTq` or `OUTr`, the agent ensures that there exists a tuple `q(X,Y)` that is true or unknown (i.e., not in `OUTq`) and a tuple `r(Y,Z)` that is true or unknown.

Aggregates are easily handled. For example, the cardinality constraint $1\{\ldots\}1$ is handled as follows: once a tuple of the relation is added into IN, all unknown tuples are added into OUT; and once all tuples but one are added into OUT, then this remaining tuple is added into IN.

4 Performance Evaluation

B-Prolog version 7.5 has been enhanced with a library of predicates for compiling ASP. Table 1 compares hand-compiled ASP2AR with Clasp (version 1.3.5) [1] and GASP [5] on CPU times for four benchmarks. ASP2AR is an order of magnitude faster than GASP. The first two programs, *p2* and *squares*, are known to require expensive grounding [5]. For them, ASP2AR considerably outperforms Clasp. The other two programs, *queens* and *color*, do not require expensive grounding, and ASP2AR is still not comparable with Clasp. Note that

Table 1. CPU time (seconds, Windows-XP, 1.4 GHz CPU, 1G RAM)

Benchmark	ASP2AR	Clasp	GASP
p2	0.27	12.29	14.90
square	0.02	32.17	0.65
queens(50)	0.98	0.156	n/a
color	3.76	0.09	>3600

the labeling strategy used has a great impact on the performance. In ASP2AR, a relation with the fewest unknown tuples is labeled first. This strategy is similar to the first-fail principle used in CLP(FD). No direct comparison was conducted with ASPeRiX since ASPeRiX does not support aggregates which are used in three of the benchmarks.

5 Conclusion

This paper has presented ASP2AR, a scheme for compiling ASP into action rules. The preliminary results show that ASP2AR is competitive for programs that require heavy grounding, and the grounding-before-solving approach is competitive for programs that require light grounding. One future project is to develop a solver that amalgamates these two approaches. Since AR is used as a common intermediate language for both CLP(FD) and ASP, it will be easy to closely integrate these two languages. Another future project is to introduce CLP(FD) features such as global constraints into ASP.

Acknowledgements

Neng-Fa Zhou is supported by NSF (No.1018006), Yi-Dong Shen is supported by NSFC (No. 60970045), and Jia-Huai You is supported by NSERC (No. 90718009).

References

1. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A User's Guide to gringo, clasp, clingo and iclingo. Technical report, University of Potsdam (2008)
2. Greco, S., Molinaro, C., Trubitsyna, I., Zumpano, E.: NP Datalog: A logic language for expressing search and optimization problems. TPLP 10(2), 125–166 (2010)
3. Lefèvre, C., Nicolas, P.: A first order forward chaining approach for answer set computing. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 196–208. Springer, Heidelberg (2009)
4. Liu, L., Pontelli, E., Son, T.C., Truszczyński, M.: Logic programs with abstract constraint atoms: The role of computations. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 286–301. Springer, Heidelberg (2007)
5. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: Answer set programming with constraints using lazy grounding. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 115–129. Springer, Heidelberg (2009)
6. Zhou, N.-F.: Programming finite-domain constraint propagators in action rules. TPLP 6(5), 483–508 (2006)

VIDEAS: A Development Tool for Answer-Set Programs Based on Model-Driven Engineering Technology[⋆]

Johannes Oetsch[1], Jörg Pührer[1], Martina Seidl[2,3],
Hans Tompits[1], and Patrick Zwickl[4]

[1] Technische Universität Wien, Institut für Informationssysteme 184/3,
Favoritenstraße 9–11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at
[2] Johannes Kepler Universität Linz, Institut für Formale Modelle und Verifikation,
Altenbergerstraße 69, A-4040 Linz, Austria
Martina.Seidl@jku.at
[3] Technische Universität Wien, Institut für Softwaretechnik, 188/3
Favoritenstraße 9–11, A-1040 Vienna, Austria
[4] FTW Forschungszentrum Telekommunikation Wien GmbH
Donau-City-Straße 1, A-1220 Vienna, Austria
zwickl@ftw.at

Abstract. In the object-oriented world, much effort is spent into the development of dedicated tools to ease programming and to prevent programming errors. Recently, the techniques of *model-driven engineering* (MDE) have been proven especially valuable to manage the complexity of modern software systems during the software development process. In the world of answer-set programming (ASP), the situation is different. Much effort is invested into the development of efficient solvers, but the pragmatics of programming itself has not received much attention and more tool support to ease the actual programming phase would be desirable. To address this issue, we introduce the tool VIDEAS which graphically supports the partial specification of answer-set programs, applying technologies provided by MDE.

Keywords: answer-set programming, model-driven engineering, ER diagrams.

1 Introduction

During the last decades, logic programming experienced a new impetus by the growth of answer-set programming (ASP) as one of the key technologies for declarative problem solving in the academic AI community. However, ASP could not attract the same interest as other programming languages outside academia so far. This lack of interest in ASP may be explained by the absence of a sufficiently supported software engineering methodology that could significantly ease the process of designing and developing ASP programs. Thus, more tool support is a declared aim of the ASP community. In particular, no modelling environment has been introduced in the context of developing answer-set

[⋆] This research has been partially supported by the the Austrian Science Fund (FWF) under grant P21698 and the Vienna Science and Technology Fund (WWTF) under grant ICT10-018.

programs that offers valuable abstraction and visualisation support during the development process. This absence of modelling tools may be explained by the fact that—in contrast to procedural programs—answer-set programs themselves are already defined at a high level of abstraction and may be regarded as executable specifications themselves. However, practice has shown that the development of answer-set programs is not always straightforward and that programs are, as all human-made artifacts, prone to errors. In fact, debugging in ASP is currently a quite active research field [1,2,3,4,5,6,7,8,9].

Consider for example the facts `airplan(boeing)` and `airplane(airbus)`. This small program excerpt already contains a mistake. A predicate name is misspelled, which might result in some unexpected program behaviour. Furthermore, most current ASP solvers do not support type checking. A notable exception is the DLV$^+$ system [10] that supports typing and concepts from object-oriented programming. If values of predicate arguments are expected to come from a specific domain only, specific constraints have to be included in the program. This requires additional programming effort and could even be a further source for programming errors.

To support answer-set programmers, we developed the tool VIDEAS, standing for "VIsual DEsign support for Answer-Set programming", which graphically supports the partial specification of answer-set programs. Due to the close relationship between answer-set programs and deductive databases, the widely used *entity relationship diagram* (ER diagram) [11] is used as a starting point for the visualisation of answer-set programs. The constraints on the problem domain from an ER diagram are automatically translated to ASP itself. Having such constraints as part of a problem encoding can be compared to using assertions in C programs. To support the development of a fact base, VIDEAS automatically generates a program providing an input mask for correctly specifying the facts. To realise VIDEAS, we used well-established technologies from the active field of *model-driven engineering* (MDE) which provides tools for building the necessary graphical modelling editors as well as the code generator.

2 Answer-Set Programming with VIDEAS

We assume basic familiarity with ASP in what follows and refer to the literature for more details [12].

In object-oriented programming as well as in data engineering, it is common to model the necessary data structures by means of graphical models like UML class diagrams (CD) or the entity relationship diagram (ER diagram) [11]. In model-driven engineering (MDE) [13], such models serve as primary development artifacts from which code can be generated. Within the development process, models are more than mere documentation items as in traditional software engineering. Besides the fact that graphical visualisation is in general easier understandable for the human software engineer and programmer, models may be automatically transformed into executable code. Consequently, inconsistencies between the models and the code are impossible.

The VIDEAS system, whose basic functionality is described in what follows, is inspired by MDE principles and intended for graphically specifying the data model of an answer-set program by means of ER diagrams. From an ER diagram, certain constraints can be automatically derived to guide the development process and to support debugging tasks. Similar approaches have been introduced in previous work [16,17] where it

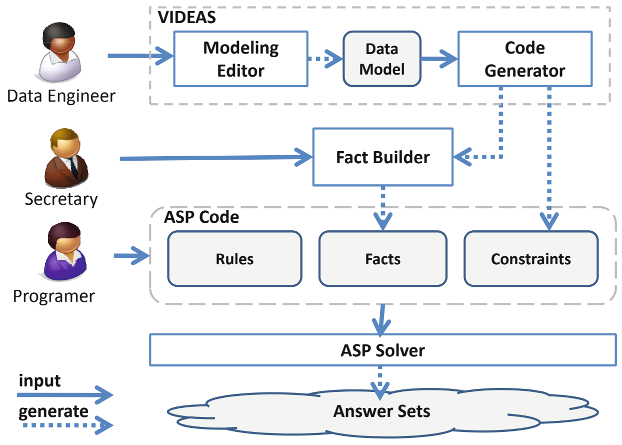


Fig. 1. The development process

is proposed to derive logic programs from *extended ER diagrams* (EER diagrams). In contrast to the VIDEAS approach, which aims at supporting the development of answer-set programs, the intention in these works was to provide a prototypical environment to experiment on various design approaches in order to reason about the instantiations of the EER diagrams. VisualASP [14] offers an environment for the graphical specification of answer-set programs by providing an editor for directly visualizing the ASP concepts. VIDEAS, in contrast, takes advantage of the abstraction power of the EER diagram and adopts the query by a diagram approach (cf. the survey article by Catarci et al. [15]) for program specification. The full potential of VIDEAS is exploited if it is integrated in a graphical development environment like the one due to Sureshkumar et al. [18].

An overview of the development process using VIDEAS is given in Fig. 1. In the VIDEAS framework, three tasks are necessary to build answer-set programs: (i) modelling, (ii) building a fact base, and (iii) implementing the program. The different tasks may be accomplished from people with different background. Specific knowledge on ASP is only required in the third step.

Modelling. In the first step, an ER diagram is specified using a graphical modelling editor that is part of the VIDEAS system (a screenshot of the editor is depicted in Fig. 2). The diagram describes entities and relations between entities of the problem domain under consideration. From the ER diagram, type and primary key constraints are derived which may be included in the final program for testing or debugging purposes. In particular, for every predicate P and each of its non-key attributes A, two rules are introduced that prohibit that two literals with predicate symbol P and different values for A sharing the same primary key are derived. Moreover, for each foreign key attribute, two rules are introduced ensuring that the key value references to an entity of the correct type. Fig. 3 presents a selection of the constraints covering the ER diagram in Fig. 2.

Building a fact base. After the modelling phase, the FactBuilder component allows to safely enter data by means of facts. The FactBuilder tool ensures that the entered data is consistent with the ER model. The resulting fact base may serve as an assertional

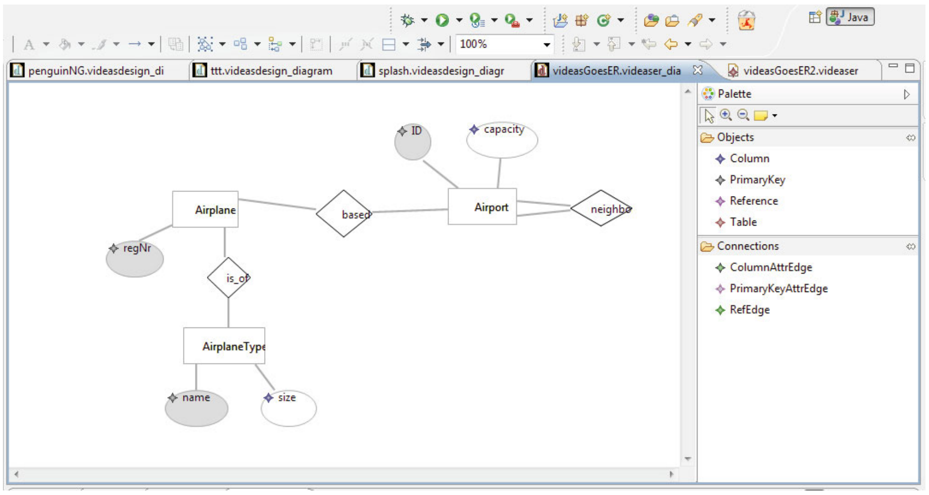


Fig. 2. Screenshot of the ER editor

```
% PRIMARY KEY CONSTRAINT
nokPkAirportCapacity(ID) :- Airport(ID,CAPACITY1),
                            Airport(ID,CAPACITY2),
                            CAPACITY1 != CAPACITY2.
:- nokPkAirportCapacity(ID), Airport(ID,CAPACITY1).

% TYPE CONSTRAINTS
okAirplaneAirplaneTypeName(NAME) :-  Airplane(_,NAME,_),
                                     AirplaneType(NAME,_).
:-  not okAirplaneAirplaneTypeName(NAME), Airplane(_,NAME,_).
okAirplaneAirportID(ID) :- Airplane(_,_,ID), Airport(ID,_).
:-  not okAirplaneAirportID(ID), Airplane(_,_,ID).
```

Fig. 3. Excerpt of constraints generated from an ER diagram

```
:add airplane
regNr: 1
airplaneType.name: Boeing737
airport.ID: ap1

% RESULTING FACT
airplane(1,Boeing737,ap1).
```

Fig. 4. An example for the FactBuilder component

knowledge base for the answer-set program. It is also possible to enter the data at a later point in time or to define multiple knowledge bases which increases the versatility of problem representations. Figure 4 gives an example exploiting the FactBuilder tool.

Implementation. Finally, the program under development has to be completed. That is, all properties of problem solutions beyond the constraints imposed by the ER diagram have to be formalised in ASP. VIDEAS does not impose any restriction on answer-set programmers concerning the implementation step but rather provides assistance for some parts of the development process by offering modelling and visualisation techniques as well as the automated generation of constraint systems.

3 Implementation

We next sketch how we developed a first prototype of VIDEAS based on standard model-engineering technologies. VIDEAS has been implemented on top of the Eclipse platform[1]. In particular, technologies provided by the Eclipse Modeling Framework (EMF)[2] and the Graphical Modeling Framework (GMF)[3] projects have been used. The meta-model representing the ER diagram modelling language has been created using the Ecore modelling language which is specified within the EMF project. Based on this Ecore model, a graphical editor has been created using GMF.

The code generator, which is implemented in Java, processes the models from the graphical editor. Again, this model is formulated in Ecore. The code generation itself can be grouped into three subsequent activities: First, the model is analysed. This allows to compute and to store the used literals based on the defined relationships, the chosen cardinalities, and the specified attributes. Second, type and primary key constraints are generated (cf. Fig. 3 for an example). Third, input forms are prompted which enable a developer to fill in values that are used for generating the facts of the program—the FactBuilder of VIDEAS (cf. Fig. 4). The FactBuilder component also implements features like the automated look-up of values from a known domain. Finally, the facts and constraints may be written to a file.

4 Conclusion and Future Work

The idea behind VIDEAS is to introduce successful techniques from model-driven engineering (MDE) to the ASP domain with the aim of supporting an answer-set programmer during a development phase. The distinguishing feature of MDE is that models are first-class citizens in the engineering process rather than mere documentation artifacts. In particular, programmers are encouraged to use ER diagrams to describe the data model of a problem domain before implementing a program. The benefit of an explicit model is that input masks for the consistent definition of a fact base for an answer-set program can be generated automatically. Furthermore, constraints represented by the

[1] `https://www.eclipse.org`
[2] `http://www.eclipse.org/modeling/emf/`
[3] `http://www.eclipse.org/modeling/gmf/`

ER model can be ported automatically into the language of ASP. Hence, consistency of any answer set with the data model of the ER diagram can always be guaranteed.

For future work, we intend to consider further concepts like inheritance relationship and other modelling languages like subsets of the UML class diagram as well. The UML class diagram may be particularly beneficial for ASP because the language-inherent extension mechanism of UML profiles may be used to adapt the UML class diagram to our specific purposes. We also plan to extend the VIDEAS framework to visualise potential inconsistencies between answer sets of a program and the data model directly at the level of the underlying ER diagram.

References

1. Brain, M., De Vos, M.: Debugging logic programs under the answer-set semantics. In: Proc. ASP 2005. CEUR Workshop Proc., pp. 141–152 (2005), CEUR-WS.org
2. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proc. NMR 2006, pp. 77–83 (2006)
3. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
4. Mikitiuk, A., Moseley, E., Truszczynski, M.: Towards debugging of answer-set programs in the language PSpb. In: Proc. ICAI 2007, pp. 635–640. CSREA Press (2007)
5. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A theoretical framework for the declarative debugging of datalog programs. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 143–159. Springer, Heidelberg (2008)
6. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Proc. AAAI 2008, pp. 448–453. AAAI Press, Menlo Park (2008)
7. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming 9(1), 1–56 (2009)
8. Wittocx, J., Vlaeminck, H., Denecker, M.: Debugging for model expansion. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 296–311. Springer, Heidelberg (2009)
9. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: On debugging non-ground answer-set programs. Theory and Practice of Logic Programming 10(4-6), 513–529 (2010)
10. Ricca, F., Leone, N.: Disjunctive logic programming with types and objects: The DLV$^+$ system. Journal of Applied Logic 5(3), 545–573 (2007)
11. Chen, P.: The entity-relationship model—Toward a unified view of data. ACM Transactions on Database Systems 1(1), 9–36 (1976)
12. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
13. Schmidt, D.C.: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
14. Febbraro, O., Reale, K., Ricca, F.: A Visual Interface for Drawing ASP Programs. In: Proc. CILC 2010 (2010)
15. Catarci, T., Costabile, M.F., Levialdi, S., Batini, C.: Visual query systems for databases: A survey. J. Visual Languages and Computing 8(2), 215–260 (1997)
16. Kehrer, N., Neumann, G.: An EER Prototyping Environment and its Implementation in a Datalog Language. In: Pernul, G., Tjoa, A.M. (eds.) ER 1992. LNCS, vol. 645, pp. 243–261. Springer, Heidelberg (1992)
17. Amalfi, M., Provetti, A.: From extended entity-relationship schemata to illustrative instances. In: Proc. LID 2008 (2008)
18. Sureshkumar, A., de Vos, M., Brain, M., Fitch, J.: APE: An AnsProlog Environment. In: Proc. SEA 2007, pp. 101–115 (2007)

The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track

Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri

Dipartimento di Matematica, Università della Calabria, Italy
aspcomp11@mat.unical.it

Abstract. Answer Set Programming is a well-established paradigm of declarative programming in close relationship with other declarative formalisms such as SAT Modulo Theories, Constraint Handling Rules, FO(.), PDDL and many others. Since its first informal editions, ASP systems are compared in the nowadays customary ASP Competition. The Third ASP Competition, as the sequel to the ASP Competitions Series held at the University of Potsdam in Germany (2006-2007) and at the University of Leuven in Belgium in 2009, took place at the University of Calabria (Italy) in the first half of 2011. Participants competed on a selected collection of declarative specifications of benchmark problems, taken from a variety of domains as well as real world applications, and instances thereof. The Competition ran on two tracks: the Model & Solve Competition, held on an open problem encoding, on an open language basis, and open to any kind of system based on a declarative specification paradigm; and the System Competition, held on the basis of fixed, public problem encodings, written in a standard ASP language. This paper briefly discuss the format and rationale of the System competition track, and preliminarily reports its results.

1 Introduction

Answer Set Programming[1] is a declarative approach to programming proposed in the area of nonmonotonic reasoning and logic programming [8,22,23,33,34,46,48]. The main advantage of ASP is its high declarative nature combined with a relatively high expressive power [16,42]; after some pioneering work [9,52], there are nowadays a number of stable and solid systems that support ASP and its variants [6,15,30,38,39,41,42,43,45,47,49,51]. Since the first informal editions (Dagstuhl 2002 and 2005), ASP systems are compared in the nowadays customary ASP Competitions [19,31], which reached now its third official edition.

The Third ASP Competition featured two tracks: the Model & Solve Competition, held on an open problem encoding, open language basis, and open to any system based on a declarative specification paradigm, and the System Competition, held on the basis

[1] For introductory material on ASP, the reader might refer to [8,24,33,46,48].

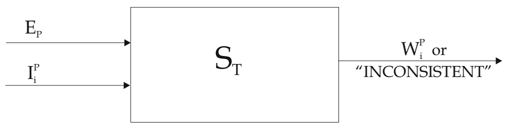


Fig. 1. Setting of the System Competition per each participant system S_T

of fixed problem encodings, written in a standard ASP language. At the time of this writing, the System Competition Track is over, while the Model & Solve one is going to start. In this paper, we focus on the former, preliminarily reporting its results and discussing the peculiarities of its format. A more detailed report, including the results of the Models & Solve track, and outcomes of non-participant systems, is under preparation.

The format of the System Competition Track (System Competition from now on) was conceived with the aim of *(i)* fostering the introduction of a standard language for ASP, and the birth of a new working group for the definition of an official standard, and *(ii)* let the competitors compare each other in fixed, predefined conditions.

Intuitively, the System Competition is run as follows (see Figure 1): a selection of problems is chosen in the open *Call for Problems* stage: for each problem P a corresponding, fixed, declarative specification E_P of P, and a number of instances $I_{P_1}, \dots, I_{P_n}$, are given. Each participant system S_T, for T a participating team, is fed with all the couples $\langle E_P, I_{P_i} \rangle$, and challenged to produce a *witness* solution to $\langle E_P, I_{P_i} \rangle$ (denoted by W_i^P) or to report that no witness exist, within a predefined amount of allowed time. A score is awarded to each S_T per each problem, based on the number of solved instances for that problem, and the time spent for solving these, as detailed in Section 6.

Importantly, problem encodings were fixed for all participants: specialized solutions on a per-problem basis were not allowed, and problems were specified in the ASP-Core language. This setting has been introduced in order to give a fair, objective measure of what one can expect when switching from a system to another, while keeping all other conditions fixed, such as the problem encoding and the default solver settings and heuristics.

The language ASP-Core collects basic ASP features common in almost every current system. A small portion of the problems have been encoded in a second language format called ASP-RfC, this latter proposed in order to encourage the standardization of other popular basic features, which still unfortunately differ (in syntax and semantics) between current systems.

ASP-Core is a conservative extension to the non-ground case of the Core language adopted in the First ASP Competition; it complies with the core language draft specified at LPNMR 2004 [2], and basically refers to the language specified in the seminal paper [34]. Most of its constructs are nowadays common for current ASP parsers. The ASP-Core language includes: ground queries, disjunctive rules with negation as failure, strong negation and arithmetic builtins; terms are constants and variables only.

The remainder of the paper is structured as follows. Section 2 discusses the Competition format and regulations. Section 3 briefly overviews the standard language adopted. In Section 6 we illustrate the scoring framework, also discussing the reasons that led to our choices. Section 4 presents the participant to the System Competition, and Section 5 the benchmark problems they competed on. Finally, in Section 8 we present and briefly discuss the actual results of the Competition.

2 System Competition Format

The regulations of the System Competition were conceived taking into account a number of aspects.

As a first observation, note that many families of formalisms, which can be considered to a large extent neighbors of the ASP community, have reached a significant level of language standardization. These range from the Constraint Handling Rules (CHR) family [1], the Satisfiability Modulo Theories SMT-LIB format [4], the Planning Domain Definition Language (PDDL) [35], to the TPTP format used in the Automated Theorem Proving System Competition (CASC) [3]. Such experiences witness that the common ground of a standard language, possibly undergoing continuous refinement and extension, has usually boosted the availability of resources, the deployment of the technology at hand into practical applications, and the effectiveness of systems. Nonetheless, ASP was still missing a standard, high-level input language.

The ASP community is nowadays mature enough for starting the development of a common standard input format. An ASP system can be roughly seen as composed of a front-end input language processor and a model generator: the first module, usually (but not necessarily) named "grounder", produces a propositional program obtained from an higher-level (non-ground) specification of the problem at hand. Incidentally, currently-developed ASP grounders have recently reached a good degree of maturity, and, above all, they have reached a fairly large degree of overlap in their input formats. This paves the way for taking the very first serious step towards the proposal of a common input language for ASP solvers. It thus makes sense to play the System Competition on the grounds of a common draft input format, in order to promote the adoption of a newly devised standard, and foster the birth of a new standardization working group. In order to met the above goals, the Competition input format has been conceived as large enough to embed all of the basic constructs included in the language originally specified in [34] (and lifted to its non-ground version), yet conservative enough to allow all the participants to adhere to the standard draft with little or no effort.

A second, but not less important aspect regards the performance of a given ASP system. Indeed, when fine-tuning on P is performed, either by improving the problem encoding or by ad-hoc tuning internal optimizations techniques, performances might greatly vary. Although it is important to encourage developers to fine-tune their systems, and then compete on this basis, on the other hand it is similarly important to put in evidence how a solver performs with a default behavior of choice: indeed, the user of an ASP system has generally little or no knowledge of the system internals, and might not be aware of which program rewritings and system optimization methods pay off in terms of performance for a given problem. The System Competition format aims at putting in evidence the performance of a system when used as an "off-the-shelf black box" on a supposedly unknown range of problem specifications. In this respect, rankings on the System Competition should give a fairly objective measure of what one can expect when switching from a system to another, while keeping all other conditions fixed (problem encoding and default solver settings). The format of the System Competition, thus, aims at measuring the performance of a given solver when used on a generic problem encoding, rather than the performance of a system fine-tuned for the specific problems selected for the current Competition.

Given the above considerations, the System Competition has been held on the basis of the principles/rules detailed next.

1. The System Competition was open to systems able to parse input written in the fixed format ASP-Core.
2. Given the selection of problems available for the Competition, the organizers have chosen, for each benchmark, a fixed representation encoded in ASP-Core, together with a set of benchmark instances. Participants have been made aware, fairly in advance, of fixed encodings, while they were provided only a small set of corresponding training instances: official ones have been kept secret until the actual Competition start. Each participant system was then launched with its default settings on all official problem instances. Scores were awarded according to the Competition scoring system (see Section 6).
3. Syntactic special-purpose solving techniques, specialized on a "per problem" basis, were forbidden. Besides such techniques, the committee classified the switch of internal solver options depending on: *(i)* command-line filenames; *(ii)* predicate and variable names; *(iii)* "signature" techniques aimed at recognizing a particular benchmark problem, such as counting the number of rules, constraints, predicates and atoms in a given encoding.

   In order to discourage the adoption of such techniques, the organizing committee kept the right to introduce syntactic means for scrambling program encodings, such as, e.g., file, predicate and variable random renaming. Furthermore, the committee kept the right, whenever necessary, to replace official program encodings with equivalent, syntactically changed versions.

   It is worth noting that, viceversa, the semantic recognition of the program structure was allowed, and even encouraged. Among allowed semantic recognition techniques, the Competition regulation explicitly included: *(i)* recognition of the class the problem encoding belongs to (e.g., stratified, positive, etc.) and possible consequent switch-on of on-purpose evaluation techniques; *(ii)* recognition of general rule and program structures (e.g., common un-stratified even and odd-cycles, common join patterns within a rule body, etc.), provided that these techniques were general and not peculiar of a given problem selected for the Competition.
4. None of the members of the organizing committee submitted a system for the competition, in order to properly play the role of neutral referee and guarantee an objective benchmark selection and rule definition process.
5. The committee did not disclose any submitted material until the end of the Competition; nonetheless, willingly participants were allowed to share their own work at any moment. In order to guarantee transparency and reproducibility of the Competition results, all participants were asked to agree that any kind of submitted material (system binaries, scripts, problems encodings, etc.) was to be made public after the Competition.

3 Competition Language Overview

The System Competition has been held over the ASP-Core language format. The language format has been introduced according to the following goals:

1. To include no less than the constructs appearing in the original A-Prolog language as formulated in [34], and be compliant with the LPNMR 2004 language draft [2].
2. To include, as an extension, a reduced number of features which are seen both as highly desirable and have now maturity for entering a standard language for ASP.
3. To appropriately choose the abovementioned extensions, in such a way that the cost of alignment of the input format would be fair enough to allow existing and future ASP solvers to comply with.
4. To have a language with non-ambiguous semantics over which widespread consensus has been reached.

According to goal 1, ASP-Core included a language with disjunctive heads and strong and negation-as-failure (NAF) negation, and did not require domain predicates; according to goals 2 and 3, ASP-Core included the notion of *query* and a few syntactic extensions. The semantic of choice for ASP-Core is the traditional answer set semantics as defined in [34], and extended to the non-ground case.

Reasonable restrictions were applied for ensuring that integers and arithmetic built-in predicates were finitely handled. Concerning disjunction, its unrestricted usage was circumscribed only to a restricted portion of the selected benchmarks. Whenever applicable (in the case of head-cycle-free [10] programs), converters to equivalent, non-disjunctive programs were made available to competitors.

The organizing committee released also a second format specification, called ASP-RfC. This latter came in the form of a ”Request for Comments” to the ASP community, and extended ASP-Core with non-ground queries, function symbols and a limited number of pre-defined aggregate functions. A couple of problems specified in ASP-RfC were selected for being demonstratively run for the System Competition by participants willing to implement the language.

The detailed ASP-Core and ASP-RfC language specification can be found at [12].

4 Participants

The System Track of the Competition featured eleven systems submitted by four teams. All systems rely on a grounding module, and can be grouped into two main classes: “native” systems, which exploit techniques purposely conceived/adapted for dealing with logic programs under stable models semantics, and “translation-based” systems, which, at some stage of the evaluation process, produce a specification in a different formalism (e.g. SAT); such specification is then fed to an external solver. The first category includes smodels, clasp and variants thereof; the second category includes Cmodels, SUP, IDP, lp2sat (based on SAT solvers) and lp2diff (based on SMT solvers). More details on teams and submitted systems follow.

Potassco. All people from this team are affiliated with the University of Potsdam, Germany. The Potassco team submitted four solvers to the System track: *clasp* [28], *claspD* [20], *claspfolio* [54], and *clasp-mt* [25]. The firstly mentioned system, clasp, features techniques from the area of Boolean constraint solving, and its primary algorithm relies on conflict-driven nogood learning. In order to deal with non-variable-free programs, it relies on the grounder *Gringo* [32]. *claspD* is an extension of clasp which is able to solve disjunctive logic programs, while *claspfolio* exploits machine-learning techniques in order to choose the best suited configuration of clasp to process the given input program. Finally, a non-competing participant is *clasp-mt*, which is a multithreaded version of clasp.

Cmodels. The team works at University of Kentucky, USA. It submitted two solvers: *Cmodels* [43] and *SUP* [44]. Cmodels can handle either disjunctive logic programs or logic programs containing choice rules; it exploits a SAT solver as a search engine for enumerating models, and also verifying model minimality whenever needed. As for SUP, it can be seen as a combination of the computational ideas behind Cmodels and smodels. Both solvers rely on the grounder Gringo.

IDP. The team counts researchers from the KRR group at K.U. Leuven, Netherlands, and participated with the system *IDP* [53]. IDP is a finite model generator for extended first-order logic theories. It implements one form of inference for FO(·), namely finite model expansion, and is based on a grounder module and a solver that searches for models of the propositional theory generated by the grounder. To cope with the System Competition format, Gringo has been used for grounding the input program.

smodels. The team works at Helsinki University of Technology, Finland. Competing systems from the group were 5: *lp2diffz3*, 3 versions of lp2sat (2gminisat, 2lminisat and 2minisat), and *smodels*. Input programs are instantiated by Gringo, then lp2diff [40] translates ASP programs into SMT specifications, so that any SMT solver supporting the QF_IDL dialect (difference logic over integers) of the SMT library can cope with the task of finding answer sets of ASP programs. lp2sat [37], on the other hand, does the same job for SAT solvers. lp2diffz3 couples lp2diff with the SMT solver *Z3* [18], whereas lp2sat-based solvers are coupled with *minisat* [21] through a series of distinct rewriting stages. Finally, smodels [51], one of the first robust ASP systems that have been made available to the community, has been included in the competition for comparison purposes given its historical importance.

5 Benchmark Suite

The benchmark suite used in the Competition has been obtained during the Call for problems stage. We selected 35 problems, out of which 20 were suitable for a proper ASP-Core problem specification. Problems can be roughly classified into the two categories of *Search* and *Query*, respectively.

Search problems require to find, if it exists, a "witness" solving the problem instance at hand, or to notify the non-existence of a witness. We herein adopt *witness* as a neutral term for denoting a solution for the problem instance at hand: in the setting of the System Competition, the notion of witness is nearly equivalent to the notion of answer set (or a subset thereof).

A query problem consists in finding all the facts (having a given signature) which hold in all the ”witnesses” of the instance for the problem at hand. In the setting of the System Competition, a query problem coincides with performing cautious reasoning over a given logic program. Problems were further classified according to their computational complexity[2] in three categories:

Polynomial problems. We classified in this category problems which are solvable in polynomial time in the size of the input data (data complexity). Such problems are usually characterized by the huge size of instance data. Although ASP systems do not aim at competing with more tailored technologies (database etc.) for solving this category of problems, several practical real-world applications fall in this category. Also, it is expected that an ASP solver can deal satisfactorily with basic skills such as answering queries over stratified recursive programs at a reasonable degree of efficiency. It is thus important to assess participants over this category. Note that, usually, polynomial problems are entirely solved by grounding modules of participant systems, with little or no effort for subsequent modules: grounders thus constitute the main technology which undergoes testing while dealing with polynomial problems. The chosen polynomial problems were: REACHABILITY, GRAMMAR-BASED INFORMATION EXTRACTION, HYDRAULIC LEAKING, HYDRAULIC PLANNING, STABLE MARRIAGE, and PARTNER UNITS POLYNOMIAL.

It is worth mentioning that, on the one hand, four problems out of six were specified in a fragment of ASP-Core having polynomial data complexity (a stratified logic program), easily solvable at grounding stage. On the other hand, both STABLE MARRIAGE and PARTNER UNITS POLYNOMIAL problems are known to be solvable in polynomial time [36,26], but have a natural declarative encoding which makes usage of disjunction. This latter encoding was used for the System Competition: thus, in these two problems, we tested the “combined” ability of grounder and solver modules. The aim was measuring if and to what extent a participant system was able to automatically converge to a polynomial evaluation strategy when fed with such a natural encoding. As a further remark, note that REACHABILITY was expressed in terms of a query problem, in which it was asked whether two given nodes were reachable in a given graph: this is a typical setting in which one can test systems on their search space tailoring techniques (such as *magic sets*) [5,7,14].

NP problems. We classified in this category NP-complete problems (or more precisely, their corresponding FNP versions) and any problem in NP not known to be polynomially solvable: these problems constitute the “core” category, in which to test the attitude of a system in efficiently dealing with problems formulated with the “Guess and Check” methodology; The chosen NP problems were: a number of puzzle-based planning problems (SOKOBANDECISION, KNIGHTTOUR, LABYRINTH, HANOITOWER, SOLITAIRE and two pure puzzle problems (MAZEGENERATION and NUMBERLINK); a classic

[2] The reader can refer to [50] for the definition of basic computational classes herein mentioned.

graph problem (GRAPHCOLOURING), a scheduling problem (DISJUNCTIVESCHEDULING) and a packing problem (PACKINGPROBLEM); eventually, two problems were related to applicative/academic settings: WEIGHT-ASSIGNMENTTREE [27] was related to the problem of finding the best join ordering in a conjunctive query, while MULTICONTEXTSYSTEMQUERYING was a query problem originating from reasoning tasks in Multi-Context Systems [17]. Noteworthy, this latter problem had an ASP-Core encoding producing several logic sub-modules, each of which having independent answer sets. The ability to efficiently handle both cross-products of answer sets and early constraint firing were thus herein assessed.

Beyond NP. We classified in this category any problem not known to be in NP/FNP. These are in general very hard problems, which, in the setting of the System Competition consisted of Σ_2^P-complete problems. Σ_2^P problems have encodings making unrestricted usage of disjunction: since a significant fraction of current systems cannot properly handle this class of problems, only STRATEGICCOMPANIES and

Table 1. 3rd ASP Competition - System Track: Benchmark List

ID	Problem Name	Contributor(s)	2nd ASP Competition	Type	Class
2	Reachability	Giorgio Terracina	Yes*	Query	P
3	Strategic Companies	Mario Alviano, Marco Maratea and Francesco Ricca	Yes*	Search	Beyond NP
6	Grammar-Based Information Extraction	Marco Manna	Yes	Search	P
10	Sokoban Decision	Wolfgang Faber	Yes	Search	NP
12	Knight Tour	Neng-Fa Zhou, Francesco Calimeri and Maria Carmela Santoro	Yes	Search	NP
13	Disjunctive Scheduling	Neng-Fa Zhou, Francesco Calimeri and Maria Carmela Santoro	Yes	Search	NP
14	Packing Problem	Neng-Fa Zhou	No	Search	NP
17	Labyrinth	Martin Gebser	Yes	Search	NP
18	Minimal Diagnosis	Martin Gebser	No	Search	Beyond NP
19	Multi Context System Querying	Peter Schueller	No	Query	NP
20	Numberlink	Naoyuki Tamura and Neng-Fa Zhou	Yes	Search	NP
22	Hanoi Tower	Miroslaw Truszczynski, Shaden Smith and Alex Westlund	Yes*	Search	NP
25	Graph Colouring	Yuliya Lierler and Marcello Balduccini	Yes	Search	NP
26	Solitaire	Yuliya Lierler and Marcello Balduccini	Yes	Search	NP
28	Weight-Assignment Tree	Yuliya Lierler	No	Search	NP
30	Hydraulic Leaking	Francesco Calimeri and Maria Carmela Santoro	Yes*	Search	P
31	Hydraulic Planning	Francesco Calimeri and Maria Carmela Santoro	Yes*	Search	P
32	Stable Marriage	Francesco Ricca, Mario Alviano and Marco Manna	No	Search	P
33	Maze Generation	Martin Brain and Mario Alviano	Yes*	Search	NP
34	Partner Units - Polynomial	Anna Ryabokon, Andreas Falkner and Gerhard Friedrich	No	Search	P

MINIMALDIAGNOSIS were selected in this category. The former is a traditional Σ_2^P problem coming from [11], while the latter originates from an application in molecular biology [29].

The list of all problems included into the System Competition benchmark suite is reported in Table 1. The presence of a star (*) in the third column means that the corresponding problem might slightly differ w.r.t. the version included into the benchmark suite of the Second ASP Competition. Detailed descriptions, and more, can be found on the Competition website [13].

6 Scoring System

For the definition of the scoring framework, we adopted as a starting point the one exploited in the first and second ASP Competitions. Such framework was mainly based on a weighted sum of the number of instances solved within the given time-bound; however, we decided to extend it by rewarding additional scores to systems well performing in terms of evaluation time.

Overview. We report below the factors that have been taken into account, together with the corresponding impact of these in the scoring system.

1. Benchmarks with many instances were intended to be prevented from dominating the overall score of a category. Thus, for a given problem P, we selected a fixed number N of instances ($N = 10$);
2. Non-sound solvers, and encodings, were strongly discouraged. Thus, if system S outputted an incorrect answer for some instance of a problem P, this invalidated *all the score* achieved by S for problem P;
3. Besides time performance, a system managing to solve a given problem instance sets a clear gap over all systems not able to. Thus, per each instance I of problem P, a flat reward has been given to a system S that correctly solved I within the allotted time;
4. Human beings are generally more receptive to the log of the changes of a value rather than to the changes themselves, especially when considering evaluation times; this is why a difference between two values is better perceived when it consists of order of magnitudes, and systems are generally perceived as clearly faster when the solving time stays orders of magnitude below the maximum allowed time. Also, systems with time performance in the same order of magnitude are perceived as comparatively similar, in terms of development effort and quality. Keeping this in mind, and analogously to what is usually done in SAT competitions[3], a logarithmically weighted bonus has been awarded to faster systems depending on the time needed for solving each instance.

Scoring Framework. A system S could get 100 points per each given benchmark problem P; then, the final score of S consists of the sum over the scores coming from all benchmarks (note that same computations have been carried out for both search and

[3] See, for instance, `http://www.satcompetition.org/2009/spec2009.html`

query problems). The overall score of a system S on a problem P counting N instances is denoted by $S(P)$. We note first that

$$S(P) = 0$$

if S returned an answer which is incorrect for at least one instance of P. Otherwise, the score has been computed by the sum

$$S(P) = S_{solve}(P) + S_{time}(P).$$

where S_{solve} is defined as

$$S_{solve}(P) = \alpha \frac{N_S}{N}$$

for N_S being the number of instances solved by P within the time limit. Actual running time has been taken into account by defining S_{time} as:

$$S_{time}(P) = \frac{100 - \alpha}{N} \sum_{i=1}^{N} \left(1 - \left(\frac{\log(t_i + 1)}{\log(t_{out} + 1)}\right)\right)$$

where t_{out} is the maximum allowed time and t_i the time spent by S while solving instance i. Both $S_{solve}(P)$ and $S_{time}(P)$ have been rounded to the nearest integer. In the System Competition t_{out} has been set to 600 seconds, $\alpha = 50$, and 10 instances were selected per each problem domain ($N = 10$).

7 Software and Hardware Settings

The Competition took place on a battery of four servers, featuring a 4 core Intel Xeon CPU X3430 running at 2.4 Ghz, with 4GB of physical RAM and PAE enabled. The operating system of choice was Linux Debian Lenny (32bit), equipped with the C/C++ compiler GCC 4.3 and common scripting/development tools. Competitors have been allowed to install their own compilers/libraries in local home directories, and to prepare system binaries for the specific Competition hardware settings. All the systems where benchmarked with just one out of four processors enabled.

Each process spawned by a participant system had access to the usual Linux process memory space (slightly less than 3GiB user space + 1GiB kernel space). The total memory allocated by all the processes was however constrained to a total of 3 GiB (1 GiB = 2^{30} bytes). The memory footprint of participant systems has been controlled by using the Benchmark Tool Run[4]. This tool is not able to detect short memory spikes (within 100 milliseconds) or, in some corner cases, memory overflow is detected with short delay: however, we pragmatically assumed the tool as the official reference.

[4] `http://fmv.jku.at/run/`

8 Results and Discussion

We briefly report in this Section the results of the competition for official participants. The overall ranking of the System Competition is reported in Table 2. Participant systems are ordered by total score, this latter awarded according to Section 6. Per each system we list three rows which report, respectively, totals, instance score (points awarded according to the S_{solve} function and proportional to the number of instances solved within time-out) and time score (points awarded according to the S_{time} function and logarithmically proportional to solving time). Columns report cumulative results for the three categories (P, NP, and Beyond-NP), and the grandtotal. Scores awarded per each benchmark problem are reported in the subsequent columns. Figure 2 reports the behavior of systems in terms of number of solved instances and solving time, per each of the three categories (2-(b), 2-(c) and 2-(d)) and on the overall (Figure 2-(a)). Timed out instances are not drawn.

Table 2. Final results

		Cumulative				P						NP												Bnd-NP	
System		Total	P	NP	Beyond NP	Reachability	Grammar-Based IE	HydraulicLeaking	HydraulicPlanning	StableMarriage	PartnerUnitsPolynomial	SokobanDecision	KnightTour	DisjunctiveScheduling	PackingProblem	Labyrinth	MCS Querying	Numberlink	HanoiTower	GraphColouring	Solitaire	Weight-AssignmentTree	MazeGeneration	StrategicCompanies	MinimalDiagnosis
claspd	**Total**	**861**	**206**	**552**	**103**	**36**	**72**	**10**	**75**	**0**	**13**	**68**	**68**	**30**	**0**	**65**	**75**	**69**	**31**	**19**	**11**	**20**	**96**	**12**	**91**
	Instance	560	145	355	60	25	50	10	50	0	10	45	40	25	0	45	50	40	25	10	10	15	50	10	50
	Time	301	61	197	43	11	22	0	25	0	3	23	28	5	0	20	25	29	6	9	1	5	46	2	41
claspfolio	**Total**	**818**	**209**	**609**	**0**	**35**	**72**	**10**	**74**	**5**	**13**	**66**	**65**	**37**	**0**	**63**	**75**	**64**	**47**	**55**	**21**	**21**	**95**	**0**	**0**
	Instance	535	150	385	0	25	50	10	50	5	10	45	35	25	0	40	50	35	35	40	15	15	50	0	0
	Time	283	59	224	0	10	22	0	24	0	3	21	30	12	0	23	25	29	12	15	6	6	45	0	0
clasp	**Total**	**810**	**213**	**597**	**0**	**36**	**72**	**10**	**75**	**6**	**14**	**78**	**63**	**38**	**0**	**78**	**75**	**65**	**39**	**23**	**21**	**21**	**96**	**0**	**0**
	Instance	520	150	370	0	25	50	10	50	5	10	50	35	25	0	50	50	35	30	15	15	15	50	0	0
	Time	290	63	227	0	11	22	0	25	1	4	28	28	13	0	28	25	30	9	8	6	6	46	0	0
idp	**Total**	**781**	**184**	**597**	**0**	**29**	**71**	**10**	**74**	**0**	**0**	**64**	**74**	**38**	**0**	**52**	**75**	**70**	**65**	**18**	**38**	**8**	**95**	**0**	**0**
	Instance	500	130	370	0	20	50	10	50	0	0	45	45	25	0	30	50	40	45	10	25	5	50	0	0
	Time	281	54	227	0	9	21	0	24	0	0	19	29	13	0	22	25	30	20	8	13	3	45	0	0
cmodels	**Total**	**766**	**184**	**510**	**72**	**29**	**71**	**10**	**74**	**0**	**0**	**67**	**56**	**21**	**0**	**62**	**75**	**30**	**51**	**29**	**18**	**6**	**95**	**0**	**72**
	Instance	510	130	335	45	20	50	10	50	0	0	45	30	20	0	45	50	20	35	20	15	5	50	0	45
	Time	256	54	175	27	9	21	0	24	0	0	22	26	1	0	17	25	10	16	9	3	1	45	0	27
lp2diffz3	**Total**	**572**	**178**	**394**	**0**	**35**	**66**	**10**	**67**	**0**	**0**	**42**	**55**	**0**	**0**	**0**	**70**	**45**	**47**	**27**	**25**	**0**	**83**	**0**	**0**
	Instance	405	135	270	0	25	50	10	50	0	0	30	35	0	0	0	50	30	35	20	20	0	50	0	0
	Time	167	43	124	0	10	16	0	17	0	0	12	20	0	0	0	20	15	12	7	5	0	33	0	0
sup	**Total**	**541**	**195**	**346**	**0**	**29**	**71**	**10**	**74**	**0**	**11**	**52**	**40**	**37**	**0**	**58**	**72**	**0**	**31**	**16**	**15**	**25**	**0**	**0**	**0**
	Instance	380	140	240	0	20	50	10	50	0	10	35	25	25	0	40	50	0	25	10	10	20	0	0	0
	Time	161	55	106	0	9	21	0	24	0	1	17	15	12	0	18	22	0	6	6	5	5	0	0	0
lp2sat2gmsat	**Total**	**495**	**185**	**310**	**0**	**30**	**66**	**10**	**68**	**0**	**11**	**36**	**10**	**32**	**0**	**46**	**71**	**22**	**47**	**17**	**29**	**-**	**0**	**0**	**0**
	Instance	365	140	225	0	20	50	10	50	0	10	30	5	25	0	35	50	15	35	10	20	-	0	0	0
	Time	130	45	85	0	10	16	0	18	0	1	6	5	7	0	11	21	7	12	7	9	-	0	0	0
lp2sat2msat	**Total**	**481**	**179**	**302**	**0**	**30**	**66**	**10**	**68**	**0**	**5**	**39**	**0**	**32**	**0**	**52**	**71**	**15**	**47**	**17**	**29**	**-**	**0**	**0**	**0**
	Instance	355	135	220	0	20	50	10	50	0	5	30	0	25	0	40	50	10	35	10	20	-	0	0	0
	Time	126	44	82	0	10	16	0	18	0	0	9	0	7	0	12	21	5	12	7	9	-	0	0	0
lp2sat2lmsat	**Total**	**472**	**171**	**301**	**0**	**28**	**66**	**10**	**67**	**0**	**0**	**35**	**0**	**32**	**0**	**53**	**71**	**17**	**47**	**17**	**29**	**-**	**0**	**0**	**0**
	Instance	350	130	220	0	20	50	10	50	0	0	30	0	25	0	40	50	10	35	10	20	-	0	0	0
	Time	122	41	81	0	8	16	0	17	0	0	5	0	7	0	13	21	7	12	7	9	-	0	0	0
smodels	**Total**	**449**	**180**	**269**	**0**	**28**	**70**	**10**	**72**	**0**	**0**	**0**	**55**	**36**	**0**	**9**	**53**	**27**	**0**	**0**	**0**	**0**	**89**	**0**	**0**
	Instance	295	130	165	0	20	50	10	50	0	0	0	30	25	0	5	35	20	0	0	0	0	50	0	0
	Time	154	50	104	0	8	20	0	22	0	0	0	25	11	0	4	18	7	0	0	0	0	39	0	0

Table 3. Final results by categories

P				NP				Beyond NP			
System	**Total Score**	**Inst. Score**	**Time Score**	**System**	**Total Score**	**Inst. Score**	**Time Score**	**System**	**Total Score**	**Inst. Score**	**Time Score**
clasp	213	150	63	claspfolio	609	385	224	claspd	103	60	43
claspfolio	209	150	59	clasp	597	370	227	cmodels	72	45	27
claspd	206	145	61	idp	597	370	227	claspfolio	0	0	0
sup	195	140	55	claspd	552	355	197	clasp	0	0	0
lp2sat2gminisat	185	140	45	cmodels	510	335	175	idp	0	0	0
cmodels	184	130	54	lp2diffz3	394	270	124	lp2diffz3	0	0	0
idp	184	130	54	sup	346	240	106	sup	0	0	0
smodels	180	130	50	lp2sat2gminisat	310	225	85	lp2sat2gminisat	0	0	0
lp2sat2minisat	179	135	44	lp2sat2minisat	302	220	82	lp2sat2minisat	0	0	0
lp2diffz3	178	135	43	lp2sat2lminisat	301	220	81	lp2sat2lminisat	0	0	0
lp2sat2lminisat	171	130	41	smodels	269	165	104	smodels	0	0	0

8.1 Overall Results

Table 2 shows claspD as the overall winner, with 861 points. 560 points were awarded for the instance score, corresponding to a total of 112 instances solved out of 200. claspfolio and clasp follow with a respective grandtotal of 818 and 810. It is worth noting that claspD is the only system, together with Cmodels, capable of dealing with the two Beyond-NP problems included in the benchmark suite, this giving to the system a clear advantage in terms of score.

8.2 Polynomial Problems

As explained in Section 5, this category mainly tests grounding pre-processing modules. All the participant systems employed Gringo 3.0.3 as grounding module: we however noticed some performance differences, both due to the different command line options fed to Gringo by participants, and by the presence of two benchmark with unstratified program encodings. The winner of the category is clasp, with 213 points, as shown in Table 3. Interestingly, Figure 2-(b) shows a sharp difference between a group of easy and hard instances: notably, these latter enforced a bigger memory footprint when evaluated. Also worth mentioning that the main cause of failure in this category has been out of memory, rather than time-out. Instances were indeed relatively large on the average.

8.3 NP Problems

The results of this category show how claspfolio (609 points) slightly outperformed clasp and idp (597 points), these latter having a slightly better time score (227 versus 224 of claspfolio).

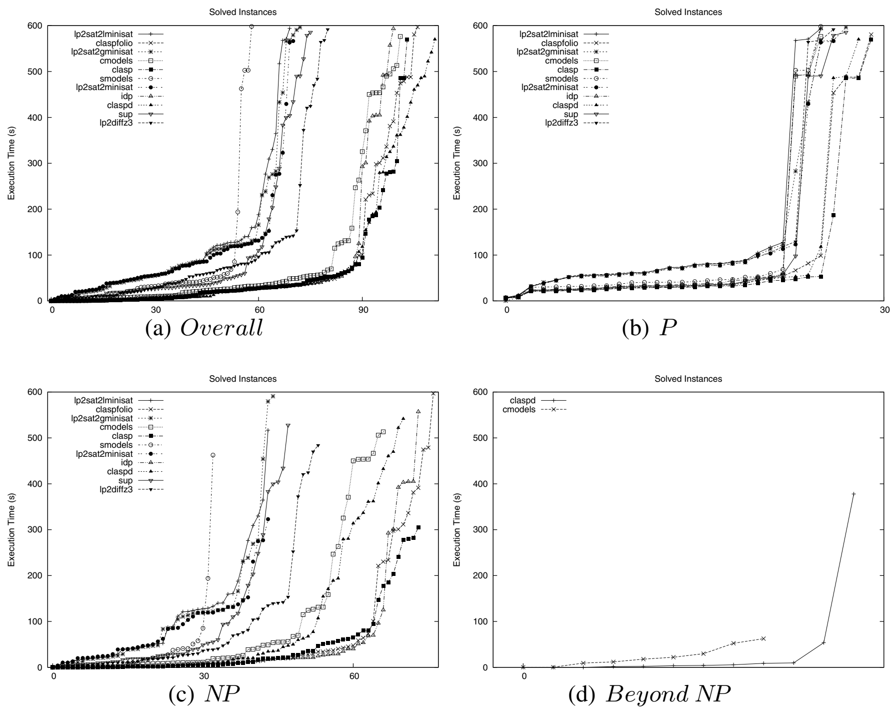


Fig. 2. Number of Solved Instances/Time graphs

8.4 Beyond-NP Problems

Only the two systems claspD and Cmodels were able to deal with the two problems in this category, for claspD able to solve and gain points on both problems and Cmodels behaving well on MINIMAL DIAGNOSIS only.

Acknowledgments

All of us feels honored of the appointment of Università della Calabria as host institution, by Marc Denecker, Mirek Truszczyński and Torsten Schaub.

Francesco Calimeri, Giovambattista Ianni and Francesco Ricca want to thank all the members of the Computer Science Group at the Dipartimento di Matematica of Università della Calabria for their invaluable collaboration, which made this event possible. A special thanks goes to all the members of the ASP and CLP communities which authored problem domains, and to participating teams, whose continuous feedback significantly helped in improving competition rules and benchmark specifications.

Francesco Calimeri, Giovambattista Ianni and Francesco Ricca want also to acknowledge Nicola Leone, as the Director of the Dipartimento di Matematica of Università della Calabria, which provided us with all the means, in form of human and technical resources, and animated earlier discussions we carried out together. A special thanks

goes to Jim Delgrande and Wolfgang Faber for their support as LPNMR-11 conference chairs and proceedings editor.

References

1. Contstraint Handling Rules, `http://dtai.cs.kuleuven.be/CHR/`
2. Core language for asp solver competitions. Minutes of the steering committee meeting at LPNMR (2004), `https://www.mat.unical.it/aspcomp2011/files/Corelang2004.pdf`
3. The CADE ATP System Competition, `http://www.cs.miami.edu/~tptp/CASC/`
4. The Satisfiability Modulo Theories Library, `http://www.smtlib.org/`
5. Alviano, M., Faber, W., Greco, G., Leone, N.: Magic sets for disjunctive datalog programs. Tech. Report 09/2009, Dipartimento di Matematica, Università della Calabria, Italy (2009), `http://www.wfaber.com/research/papers/TRMAT092009.pdf`
6. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ Approach to Answer Set Solving. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 95–109. Springer, Heidelberg (2005)
7. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: PODS 1986, Cambridge, Massachusetts, pp. 1–15 (1986)
8. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. In: CUP (2003)
9. Bell, C., Nerode, A., Ng, R.T., Subrahmanian, V.S.: Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. JACM 41, 1178–1215 (1994)
10. Ben-Eliyahu-Zohary, R., Palopoli, L.: Reasoning with Minimal Models: Efficient Algorithms and Applications. AI 96, 421–449 (1997)
11. Cadoli, M., Eiter, T., Gottlob, G.: Default Logic as a Query Language. IEEE TKDE 9(3), 448–463 (1997)
12. Calimeri, F., Ianni, G., Ricca, F.: Third ASP Competition, File and language formats (2011), `http://www.mat.unical.it/aspcomp2011/files/LanguageSpecifications.pdf`
13. Calimeri, F., Ianni, G., Ricca, F., The Università della Calabria Organizing Committee: The Third Answer Set Programming Competition homepage (2011), `http://www.mat.unical.it/aspcomp2011/`
14. Cumbo, C., Faber, W., Greco, G.: Improving Query Optimization for Disjunctive Datalog. In: Proceedings of the Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003, pp. 252–262 (2003)
15. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: GASP: Answer set programming with lazy grounding. FI 96(3), 297–322 (2009)
16. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys 33(3), 374–425 (2001)
17. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed Nonmonotonic Multi-Context Systems. In: 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010), Toronto, Canada, May 9-13, pp. 60–70. AAAI Press, Menlo Park (2010)
18. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
19. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)

20. Drescher, C., Gebser, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008), pp. 422–432. AAAI Press, Sydney (2008)
21. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
22. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In: Logic-Based Artificial Intelligence, pp. 79–103 (2000)
23. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS 22(3), 364–418 (1997)
24. Eiter, T., Ianni, G., Krennwallner, T.: Answer Set Programming: A Primer. In: Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School - Tutorial Lectures, Brixen-Bressanone, Italy, August 2009, pp. 40–110 (2009)
25. Ellguth, E., Gebser, M., Gusowski, M., Kaufmann, B., Kaminski, R., Liske, S., Schaub, T., Schneidenbach, L., Schnor, B.: A simple distributed conflict-driven answer set solver. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 490–495. Springer, Heidelberg (2009)
26. Falkner, A., Haselböck, A., Schenner, G.: Modeling Technical Product Configuration Problems. In: Proceedings of ECAI 2010 Workshop on Configuration, Lisbon, Portugal, pp. 40–46 (2010)
27. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database System Implementation. Prentice-Hall, Englewood Cliffs (2000)
28. Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver *clasp*: Progress report. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 509–514. Springer, Heidelberg (2009)
29. Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting Inconsistencies in Large Biological Networks with Answer Set Programming. TPLP 11(2), 1–38 (2011)
30. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 386–392 (2007)
31. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 3–17. Springer, Heidelberg (2007)
32. Gebser, M., Schaub, T., Thiele, S.: grinGo: A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
33. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective. AI 138(1-2), 3–38 (2002)
34. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC 9, 365–385 (1991)
35. Gerevini, A., Long, D.: Plan constraints and preferences in PDDL3 - the language of the fifth international planning competition. Technical report (2005), `http://cs-www.cs.yale.edu/homes/dvm/papers/pddl-ipc5.pdf`
36. Gusfield, D., Irving, R.W.: The stable marriage problem: structure and algorithms. MIT Press, Cambridge (1989)
37. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics 16(1-2), 35–86 (2006)
38. Janhunen, T., Niemelä, I.: GNT — A solver for disjunctive logic programs. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 331–335. Springer, Heidelberg (2003)
39. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.-H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. ACM TOCL 7(1), 1–37 (2006)

40. Janhunen, T., Niemelä, I., Sevalnev, M.: Computing Stable Models via Reductions to Difference Logic. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 142–154. Springer, Heidelberg (2009), doi:10.1007/978-3-642-04238-6_14
41. Lefèvre, C., Nicolas, P.: The first version of a new ASP solver: `aSPeRiX`. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 522–527. Springer, Heidelberg (2009)
42. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL 7(3), 499–562 (2006)
43. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 447–451. Springer, Heidelberg (2005)
44. Lierler, Y.: Abstract Answer Set Solvers. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 377–391. Springer, Heidelberg (2008)
45. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 346–350. Springer, Heidelberg (2003)
46. Lifschitz, V.: Answer Set Planning. In: ICLP 1999, Las Cruces, New, Mexico, USA, pp. 23–37 (1999)
47. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. AI 157(1-2), 115–137 (2004)
48. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm – A 25-Year Perspective, pp. 375–398 (1999)
49. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A System for Answer Set Programming. In: Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, NMR 2000 (2000), `http://xxx.lanl.gov/abs/cs/0003033v1`
50. Papadimitriou, C.: Computational Complexity. Addison-Wesley, Reading (1994)
51. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI 138, 181–234 (2002)
52. Subrahmanian, V.S., Nau, D., Vago, C.: WFS + Branch and Bound = Stable Models. IEEE TKDE 7(3), 362–377 (1995)
53. Wittocx, J., Mariën, M., Denecker, M.: The IDP system: a model expansion system for an extension of classical logic. In: Logic and Search, Computation of Structures from Declarative Descriptions, LaSh 2008, Leuven, Belgium, pp. 153–165 (2008)
54. Ziller, S., Gebser, M., Kaufmann, B., Schaub, T.: An Introduction to `claspfolio`. Institute of Computer Science, University of Potsdam, Germany (2010), `http://www.cs.uni-potsdam.de/claspfolio/manual.pdf`

Author Index